

**Wydział Elektroniki i Technik
Informacyjnych
Politechnika Warszawska**

Systemy Mikroprocesorowe w Sterowaniu
Ćwiczenia laboratoryjne

Patryk Chaber, Andrzej Wojtulewicz

Warszawa 2022

Spis treści

1. Wstęp	4
2. Ćwiczenie 1: Podstawy pracy z zestawem uruchomieniowym	5
2.1. Wprowadzenie	5
2.2. Treść ćwiczenia	5
2.2.1. Tworzenie pierwszego projektu w STM32CubeIDE	5
2.2.2. STM32CubeIDE	5
2.2.3. Podłączenie mikrokontrolera	6
2.2.4. Stworzenie projektu	9
2.2.5. Wgranie programu na mikrokontroler	16
2.2.6. Konfiguracja sprzętowa mikrokontrolera	23
2.2.7. Konfiguracja zegarów mikrokontrolera	24
2.2.8. Odczyt wartości cyfrowej	27
2.2.9. Obsługa alfanumerycznego wyświetlacza LCD	28
2.2.10. Konfiguracja PWM	30
2.2.11. Odczyt i wykorzystanie wejścia analogowego	32
2.3. Wykonanie ćwiczenia	34
3. Ćwiczenie 2: obsługa prostych czujników i urządzeń wykonawczych mikroprocesorowego systemu automatyki przy wykorzystaniu systemu przerwań	36
3.1. Wprowadzenie	36
3.2. Treść ćwiczenia	36
3.2.1. Przerwania	36
3.2.2. Przerwania zewnętrzne	38
3.2.3. Timery	40
3.2.4. Przebieg laboratorium	42
3.2.5. Opóźnienie	43
3.2.6. Ustawienie podziału priorytetów przerwań	45
3.2.7. Odmierzanie czasu przy użyciu timera ogólnego przeznaczenia	45
3.2.8. Niwelowanie drgań styków	46
3.2.9. Obsługa klawiatury numerycznej	49
3.2.10. Okresowe wykonywanie pomiarów	52
3.2.11. Dodatkowe informacje	53
3.3. Wykonanie ćwiczenia	54
4. Ćwiczenie 3: Obsługa złożonych czujników i urządzeń wykonawczych mikroprocesorowego systemu automatyki	55
4.1. Wprowadzenie	55
4.2. Treść ćwiczenia	55
4.2.1. Pętla prądowa 4-20 mA	55
4.2.2. Cyfrowy przetwornik temperatury z termometrem rezystancyjnym	56
4.2.3. Transmisja szeregową	58
4.2.4. Implementacja na ZL27ARM	59
4.2.5. Transmisja szeregową – standard MODBUS RTU	61

4.2.6.	Stanowisko grzejąco-chłodzące	65
4.3.	Wykonanie ćwiczenia	67
5.	Ćwiczenie 4: Obsługa wyświetlacza graficznego LCD (panelu dotykowego), wykorzystanie jednostki zmiennopozycyjnej do przetwarzania sygnałów . .	69
5.1.	Wprowadzenie	69
5.2.	Treść ćwiczenia	69
5.2.1.	STM32F746G-DISCO	69
5.2.2.	Biblioteka HAL	70
5.2.3.	Wyświetlacz LCD z ekranem dotykowym	71
5.2.4.	Jednostka do obliczeń zmiennoprzecinkowych	72
5.2.5.	Szablon projektu	72
5.2.6.	Fraktal	77
5.3.	Wykonanie ćwiczenia	79
6.	Projekt 1: Implementacja algorytmów regulacji PID i DMC prostego procesu dynamicznego, interfejs użytkownika, archiwizacja pomiarów, dobór nastaw algorytmów, badania porównawcze	80
6.1.	Wprowadzenie	80
6.2.	Treść projektu	80
6.2.1.	Zadanie regulacji	80
6.2.2.	Algorytm PID	84
6.2.3.	Reguły Zieglera-Nicholsa	90
6.2.4.	Metoda „inżynierska”	90
6.2.5.	Algorytm DMC	91
6.2.6.	Symulowany obiekt	94
6.2.7.	Płytki z przetwornikami	94
6.2.8.	Komunikacja na przykładzie pozyskiwania odpowiedzi skokowej	95
6.2.9.	Komunikacja na przykładzie nieustannego odczytu pomiarów	97
6.2.10.	Zadanie do wykonania	98
6.3.	Wykonanie projektu	99
7.	Projekt 2: Identyfikacja modeli (typu odpowiedzi skokowej) procesu laboratoryjnego, implementacja algorytmu regulacji DMC, dobór nastaw algorytmu, badania porównawcze	100
7.1.	Wprowadzenie	100
7.2.	Treść projektu	100
7.2.1.	Stanowisko grzejąco-chłodzące	100
7.2.2.	Stany awaryjne	100
7.2.3.	Wizualizacja	102
7.2.4.	DMC	102
7.2.5.	Ingerencja operatora	102
7.3.	Wykonanie projektu	103
8.	Projekt 3: System operacyjny czasu rzeczywistego	105
8.1.	Wprowadzenie	105
8.2.	Treść projektu	105
8.2.1.	System operacyjny czasu rzeczywistego	105
8.2.2.	Implementacja	106
8.2.3.	Zadanie do wykonania	108
8.3.	Wykonanie projektu	109
8.3.1.	Ocenianie	110
8.3.2.	Uwagi	110

1. Wstęp

2. Ćwiczenie 1: Podstawy pracy z zestawem uruchomieniowym

2.1. Wprowadzenie

Celem pierwszego ćwiczenia jest zapoznanie studenta z obsługą środowiska programistycznego STM32CubeIDE oraz nauka podstawowej obsługi mikrokontrolera. Uwaga w ramach tego przedmiotu skupiać się będzie na mikrokontrolerach z rodziny STM32, lecz przedstawiane dalej koncepcje są obecne w analogicznej formie w innych komputerach jednokładowych. Na potrzeby tego ćwiczenia wykorzystany zostanie mikrokontroler STM32F103VBT6 będący częścią płytki rozwojowej o nazwie ZL27ARM.

W kolejnych sekcjach tego rozdziału student zapozna się z zestawem uruchomieniowym, procesem przygotowywania i uruchomienia prostych programów uwzględniających obsługę portów wejścia-wyjścia, obsługę wyświetlacza tekstowego LCD, sterowanie szerokością impulsu oraz przetwornik analogowo-cyfrowy.

2.2. Treść ćwiczenia

2.2.1. Tworzenie pierwszego projektu w STM32CubeIDE

Projekt, który zostanie wykonany w ramach tego ćwiczenia laboratoryjnego, będzie nakierowany na wspomniany wcześniej mikrokontroler STM32F103VBT6. Językiem programowania wykorzystanym do implementacji kolejnych funkcjonalności będzie język C, choć równie dobrze można wykorzystać w tym celu C++. Warto zwrócić uwagę, że obecnie ekosystem STM32Cube skupia się na wykorzystaniu biblioteki o nazwie HAL (ang. Hardware Abstraction Layer), która to będzie wykorzystywana dalej w tym skrypcie. Na potrzeby jednak początkowych programów wykorzystana zostanie biblioteka SPL (ang. Standard Peripheral Library), która jest zestawem funkcji i struktur ułatwiających pracę z mikrokontrolerami z rodziny STM32. Dostępność tych bibliotek i w szczególności ich rozwój jest coraz bardziej ograniczony, nie mniej pozwalają na większą kontrolę nad działaniem mikrokontrolera, nie wprowadzając dodatkowej warstwy abstrakcji, która może wprowadzać zamieszanie w początkowych projektach. Biblioteka SPL dostarczona została przez prowadzącego laboratorium.

2.2.2. STM32CubeIDE

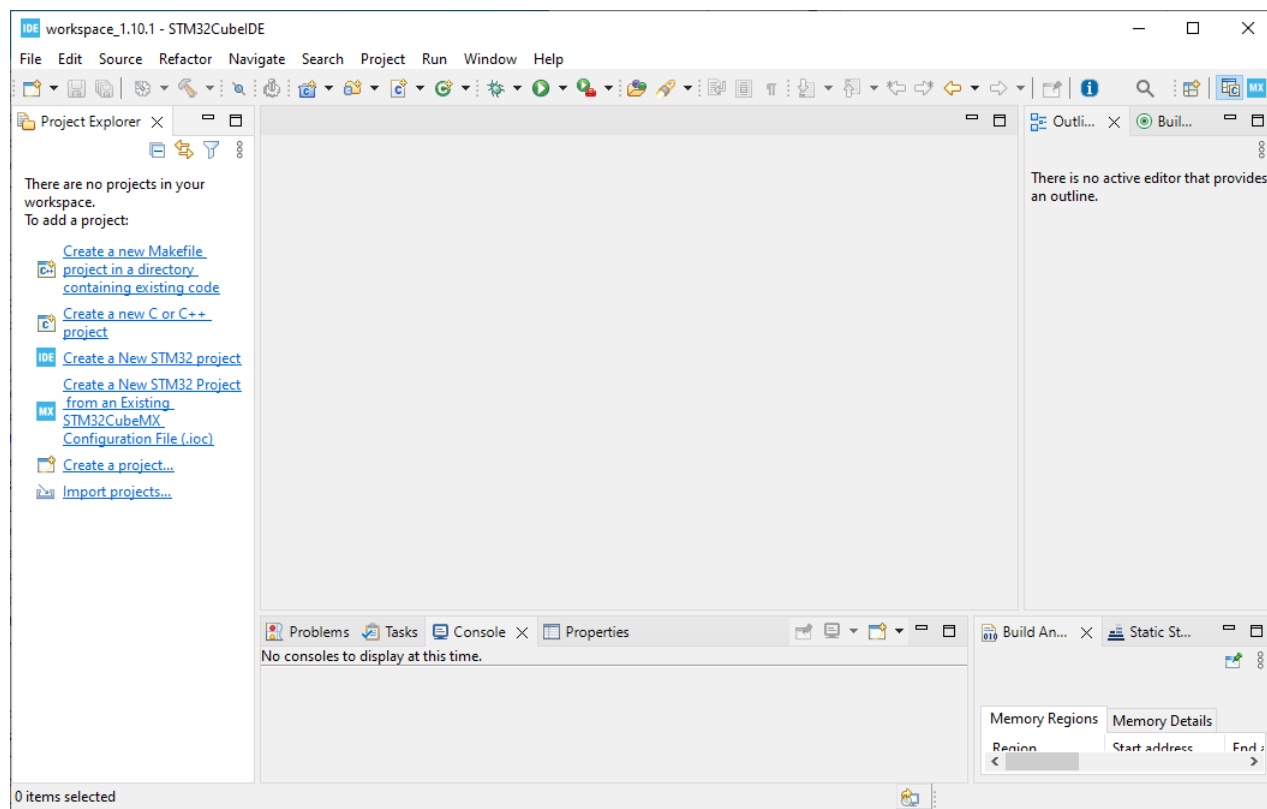
Zintegrowane środowisko deweloperskie o nazwie STM32CubeIDE bazuje na oprogramowaniu Eclipse IDE. Powoduje to, że często w trakcie korzystania z tego oprogramowania może pojawiać się odniesienie do nomenklatury związanej z tym środowiskiem, takie jak „perspektywa” czy „katalog roboczy”. Perspektywa, w kontekście tego oprogramowania, oznacza układ oraz rodzaj widoków w widocznym oknie edytora. W przypadku gdyby STM32CubeIDE wyświetlałoby zapytanie o zmianę perspektywy, w znacznej większości

przypadków warto się na to zgodzić, zaznaczając dodatkowo, aby zgoda ta dotyczyła także wszystkich kolejnych zapytań. Katalog roboczy, jest to katalog w którym składowane będą wszystkie pliki związane z projektami (poza tymi, które wprost zostały dodane wyłącznie jako dowiązania). Reszta pojęć związana z oprogramowaniem STM32CubeIDE będzie wyjaśniana wraz z dalszym poznawaniem tego środowiska.

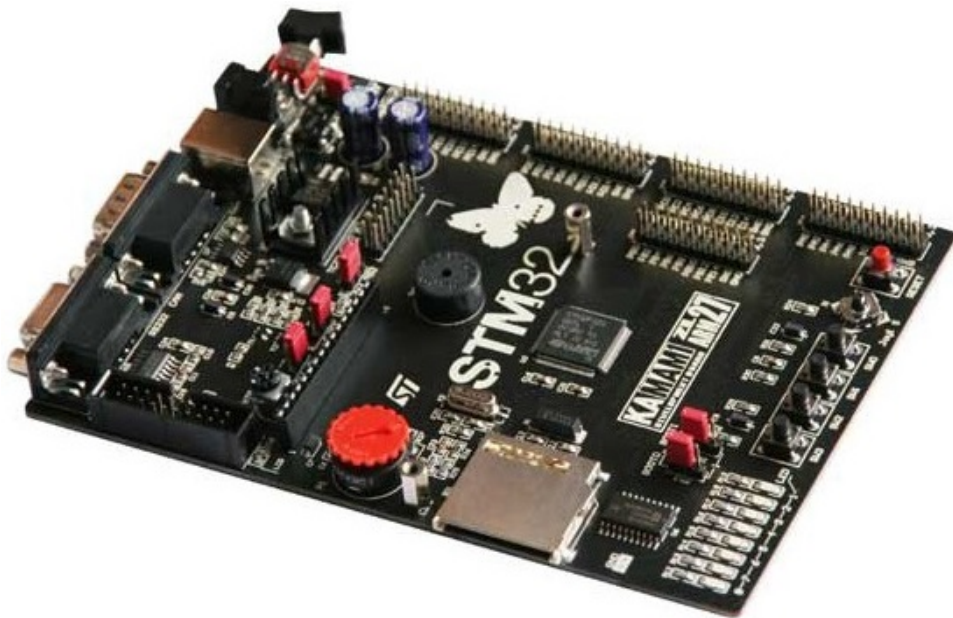
Po uruchomieniu oprogramowania STM32CubeIDE pojawić się powinno okno, jak na rys. 2.1. W tym oknie widoczne są przede wszystkim: widok drzewa projektów (*Project Explorer* po lewej stronie IDE), widok edycji poszczególnych plików (centralna część ekranu), widok wyjścia kompilacji (dolna część okna) oraz pasek narzędzi znajdujący się w górnej części okna – jego zawartość zależy od obecnie aktywnej perspektywy. Często przy pierwszym uruchomieniu STM32CubeIDE zobaczyć można także perspektywę powitalną (w razie gdyby nie była widoczna można ją odnaleźć w menu *Help*→*Information Center*).

2.2.3. Podłączenie mikrokontrolera

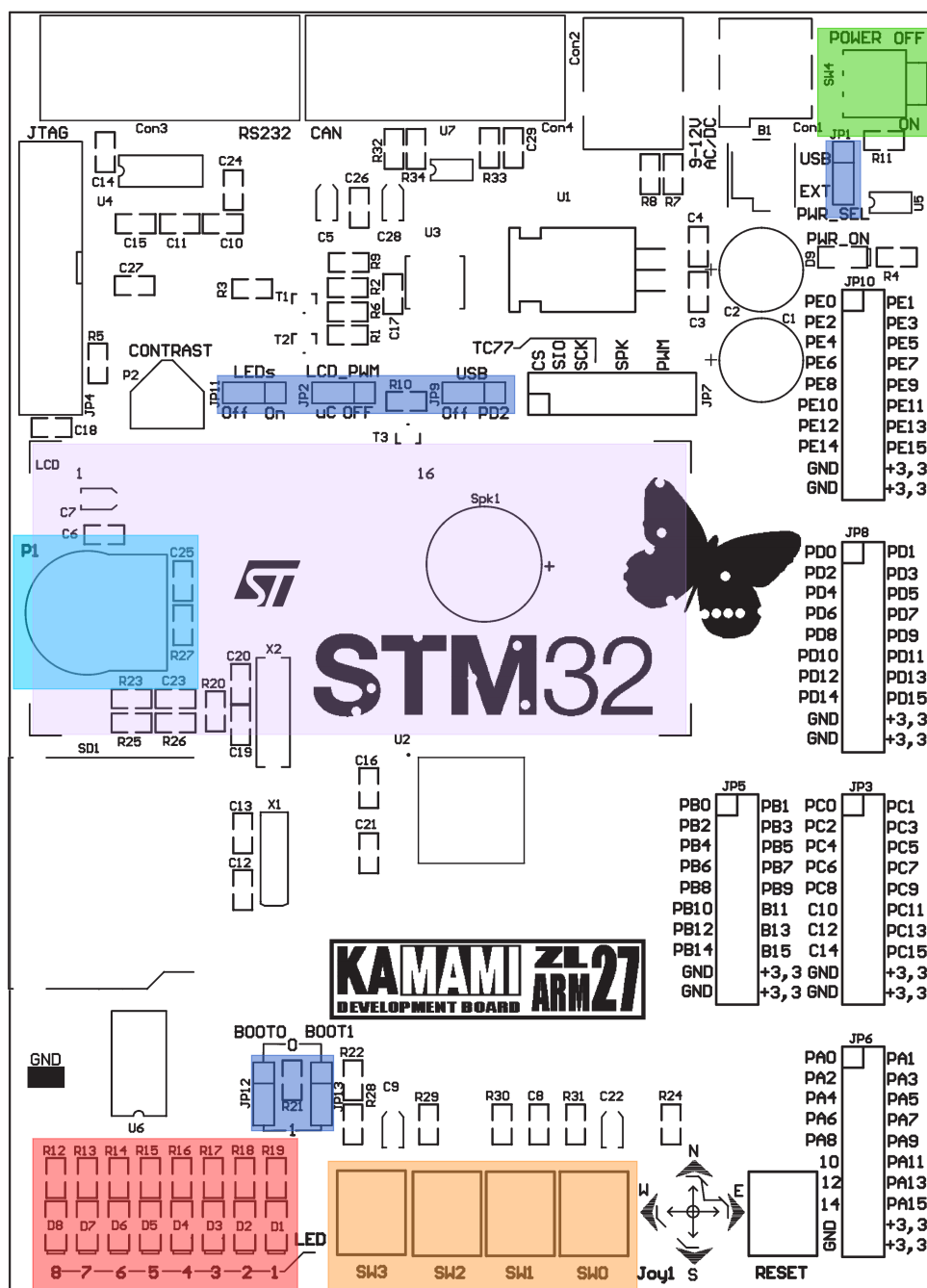
Jako mikrokontroler użyty zostanie STM32F103VBT6, zawierający się w zestawie uruchomieniowym ZL27ARM (rys. 2.2). Ogólne informacje dotyczące tego zestawu znajdują się w Instrukcji Użytkownika związanej z tym zestawem (tam też znajduje się schemat płytki, który jest widoczny na rys. 2.3), natomiast szczegółowe informacje na temat samego mikrokontrolera STM32F103VB można znaleźć w dokumentacji znajdującej się na stronie producenta (wartymi szczególnej uwagi są pliki Datasheet oraz RM0008). Programowanie zestawu ZL27ARM odbywać się będzie za pośrednictwem programatora *J-LINK* (firmy *SEGGER*) w wersji edukacyjnej (*EDU*) – rys. 2.4. Jest on podłączany poprzez złącze *JTAG* (*Joint Test Action Group*), które pozwala na testowanie (w tym debugowanie



Rys. 2.1. Okno środowiska STM32CubeIDE



Rys. 2.2. Płytką rozwojowa ZL27ARM z mikrokontrolerem STM32F103VBT6



Rys. 2.3. Schemat płytki ZL27ARM z zaznaczonymi kluczowymi elementami płytki: zworki konfiguracyjne (niebieski), przełącznik zasilania (zielony), wyświetlacz LCD (jasny fioletowy), potencjometr (błękitny), przyciski (pomarańczowy) oraz diody LED (czerwony)

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 umiejscowione 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.

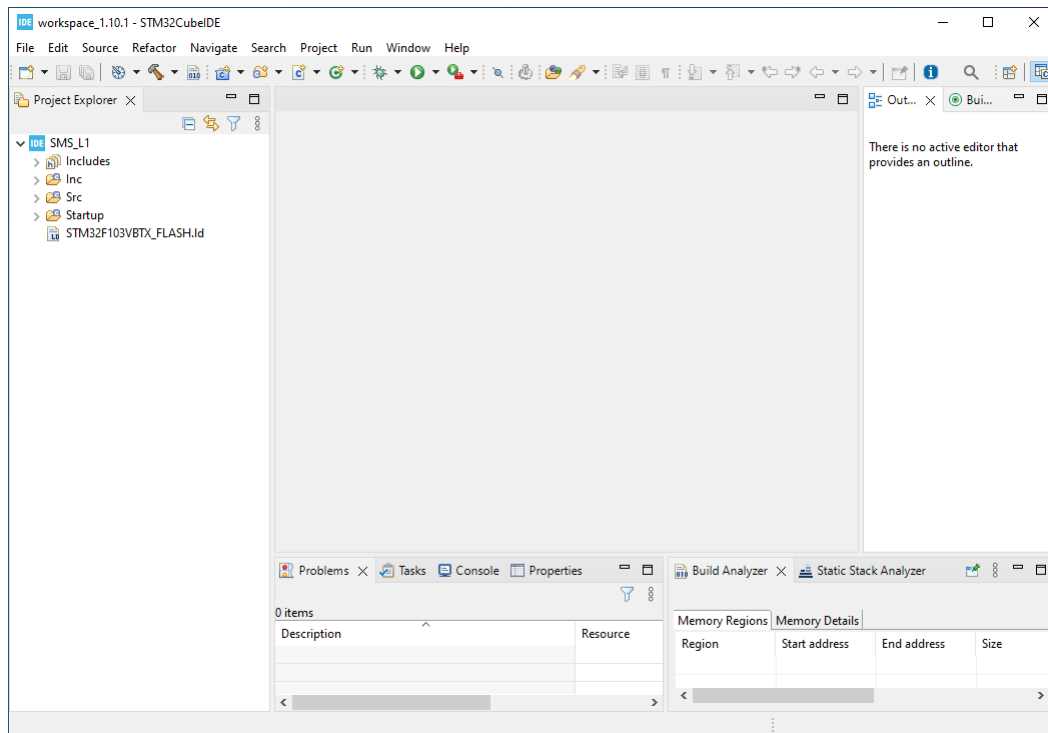
2.2.4. Stworzenie projektu

Aby stworzyć pierwszy projekt należy w oprogramowaniu STM32CubeIDE wybrać menu *File* → *New* → *STM32 Project*. Następnie w oknie, które się pojawiło, w polu *Commercial Part Number* należy wpisać nazwę mikrokontrolera, który rozważamy, tj. STM32F103VBT6. W widoku z prawej strony u dołu pojawi się lista mikrokontrolerów spełniających nasze wymagania (rys. 2.6). Należy wybrać odpowiedni i wcisnąć przycisk *Next*. W kolejnym oknie (rys. 2.7) należy wybrać nazwę projektu – w ramach tego ćwiczenia wybierzemy SMS_L1. Dodatkowo, w opcjach generacji kodu (*Targeted Project Type*), należy wybrać brak generacji kodu, tj. zaznaczyć opcję *Empty*, aby nie zostały dołączone domyślnie pliki biblioteki HAL. Po wciśnięciu przycisku *Finish*, w lewym widoku, tj. widoku drzew projektów powinien pojawić się nowy projekt o nazwie właśnie SMS_L1, tak jak jest to widoczne na rys. 2.5.

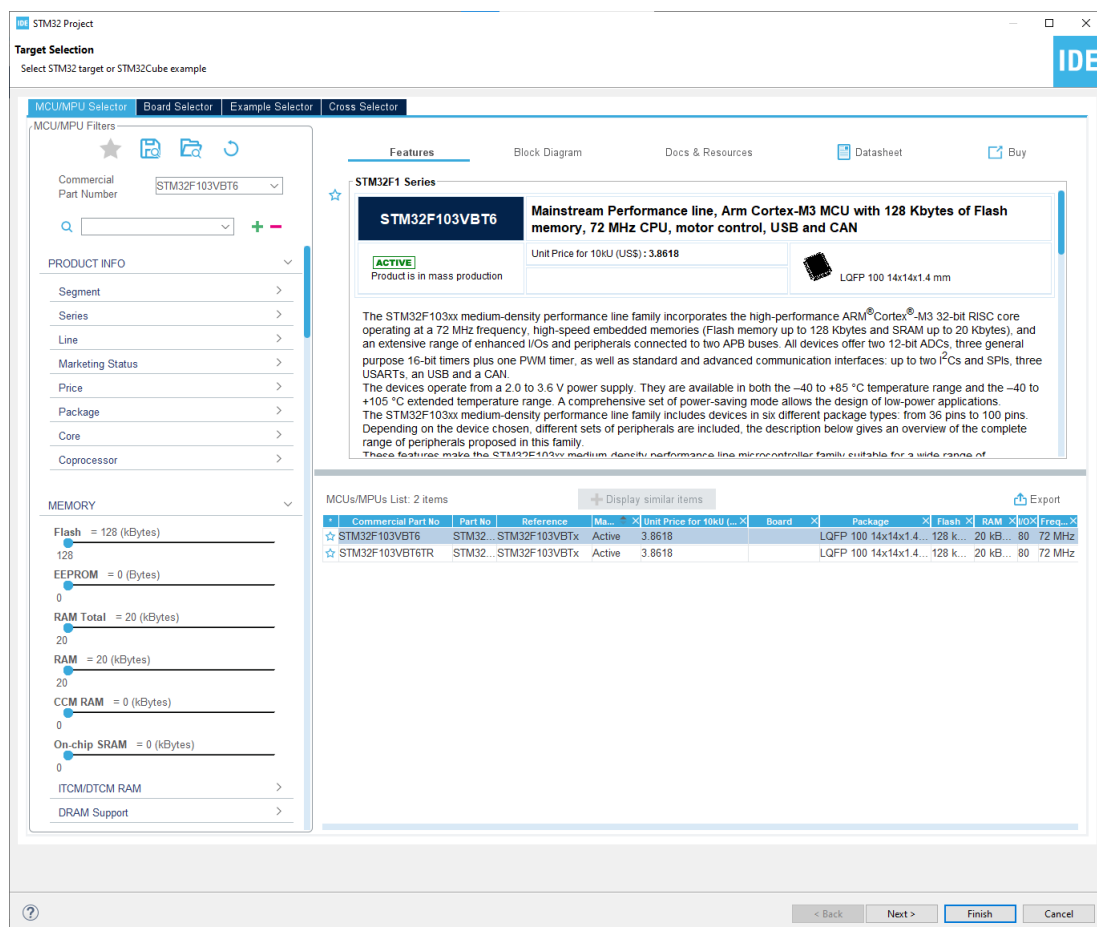
Obecnie przygotowany kod pozwala na tworzenie projektów z wykorzystaniem pisania po rejestrach. Jest to metoda pisania oprogramowania na mikrokontrolery rekomendowana przez wielu programistów, jako, że zapewnia niemal absolutną kontrolę nad wykonaniem programu. W ramach tych ćwiczeń nie będziemy skupiali się na tym podejściu, ponieważ zakłada ono doskonałą znajomość dokumentacji rozważanego mikrokontrolera, natomiast gotowy kod jest nieprzenoszalny.



Rys. 2.4. Programator firmy SEGGER, *J-LINK EDU*

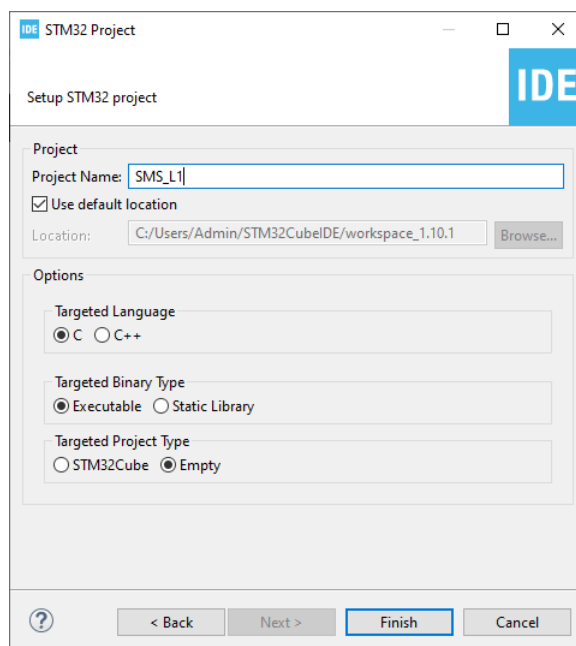


Rys. 2.5. Okno środowiska STM32CubeIDE bezpośrednio po stworzeniu pierwszego projektu

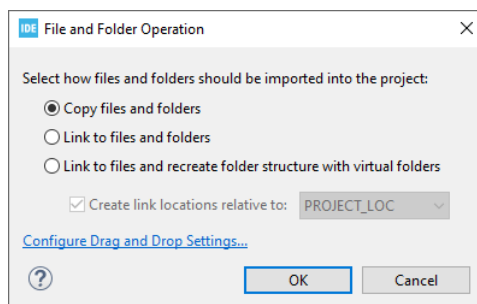


Rys. 2.6. Okno wyboru platformy docelowej (Target Selection)

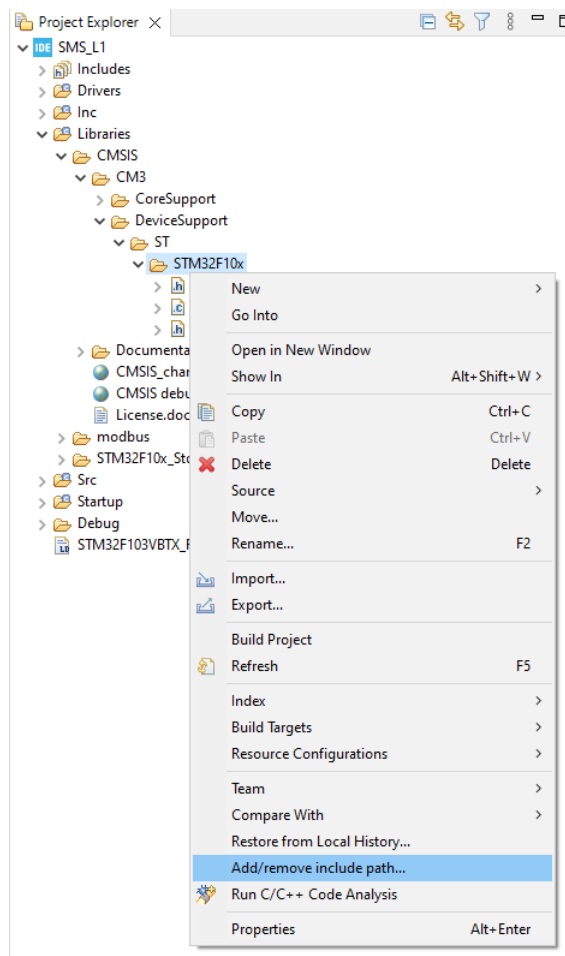
2.2. TREŚĆ ĆWICZENIA



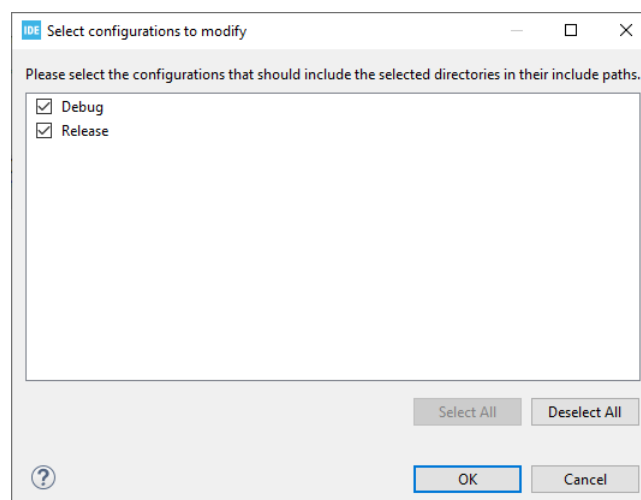
Rys. 2.7. Okno podstawowej konfiguracji projektu



Rys. 2.8. Okno wyboru sposobu importowania katalogów z plikami do projektu



Rys. 2.9. Menu kontekstowe pozwalające na dodanie katalogu projektu do listy ścieżek z plikami nagłówkowymi



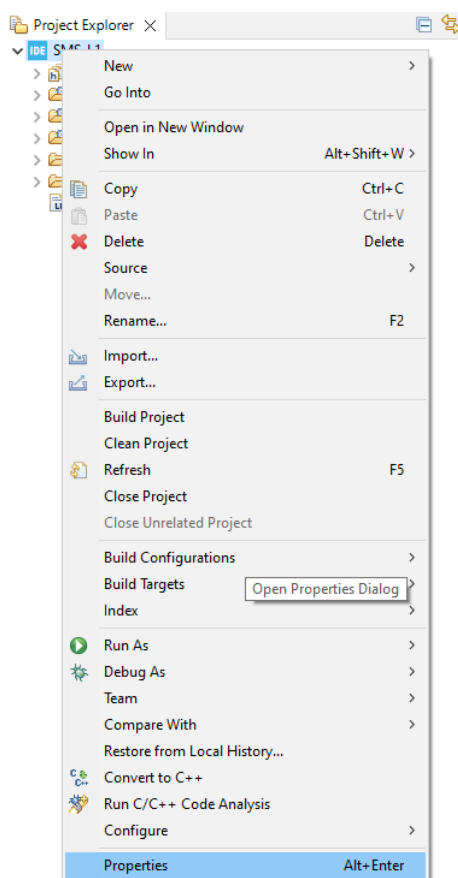
Rys. 2.10. Okno wyboru konfiguracji dla której pliki nagłówkowe mają być dodane

2.2. TREŚĆ ĆWICZENIA

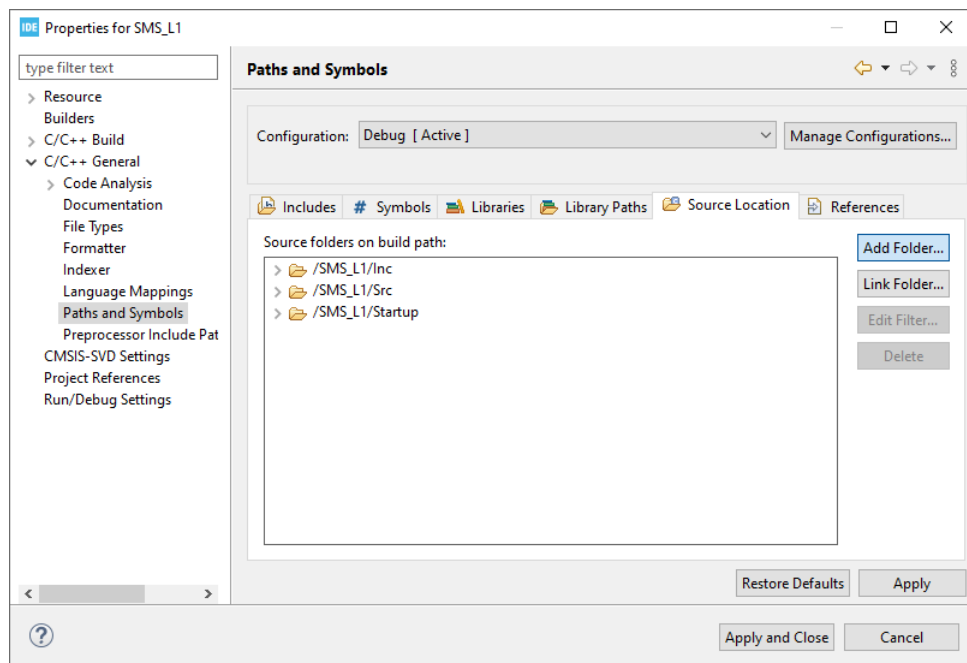
Korzystanie z bibliotek SPL wymaga kilku dodatkowych zabiegów. Jako pierwsze należy dodać bibliotekę SPL i przy okazji warto wyposażyć się w podstawowe sterowniki, m.in. do wyświetlacza LCD1602. W tym celu z katalogu wskazanego przez prowadzącego należy przeciągnąć metodą „przeciągnij i upuść” katalogi o nazwie *Drivers* oraz *Libraries* na korzeń projektu, aby trafiły do głównego katalogu projektu. Po przeciągnięciu ukazuje się okno (rys. 2.8) z pytaniem czy skopiować pliki czy wyłącznie je dołączyć, w którym należy wybrać kopiowanie (pierwsza opcja) i kliknąć przycisk *OK*. Następnie należy wskazać kompilatorowi, że w tych nowych katalogach znajdują się przydatne pliki. W tym celu należy rozwinąć poddrzewo katalogu *Drivers*, kliknąć prawym przyciskiem myszy na katalog *LCD1602* i z menu kontekstowego wybrać *Add/remove include path...* (rys. 2.9), a w oknie, które się pojawiło (rys. 2.10) kliknąć *OK*. W ten sposób dodany został ten katalog do listy ścieżek, gdzie kompilator będzie szukał plików nagłówkowych gdy wykonywana będzie kompilacja w konfiguracji zarówno *Debug*, jak i *Release* (ramach tego skryptu rozważać będziemy jednak wyłącznie konfigurację *Debug*). Czynność tę należy wykonać także dla katalogów:

- Libraries/CMSIS/CM3/CoreSupport
- Libraries/CMSIS/CM3/DeviceSupport/ST/STM32F10x
- Libraries/STM32F10x_StdPeriph_Driver/inc

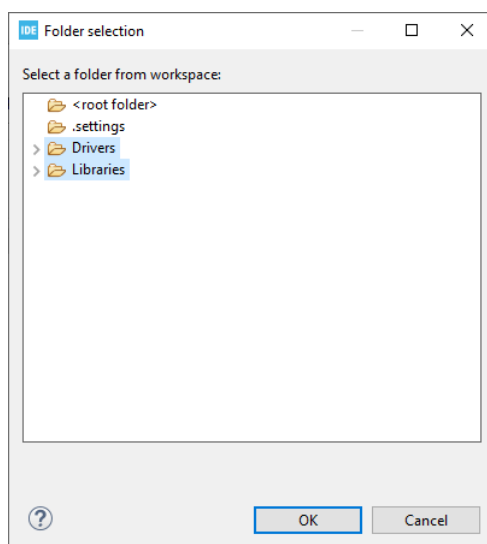
Następnie należy wskazać kompilatorowi, że w dodanych katalogach znajdują się pliki, które powinny zostać skompilowane. W tym celu należy kliknąć prawym przyciskiem my-



Rys. 2.11. Menu kontekstowe projektu pozwalające na otwarcie szczegółowych ustawień projektu



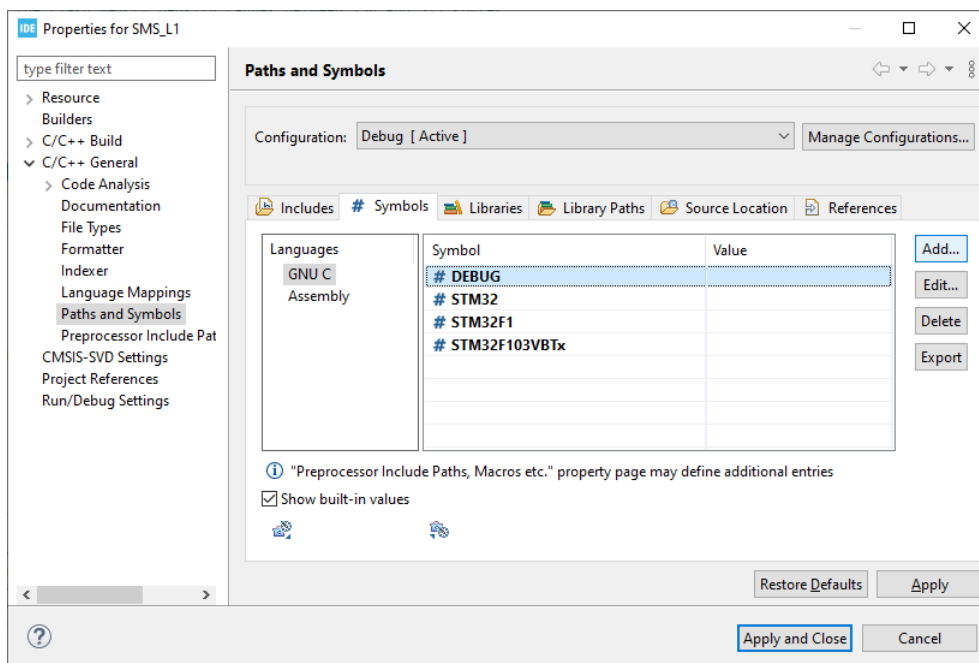
Rys. 2.12. Okno szczegółowych ustawień projektu – ścieżki z plikami źródłowymi



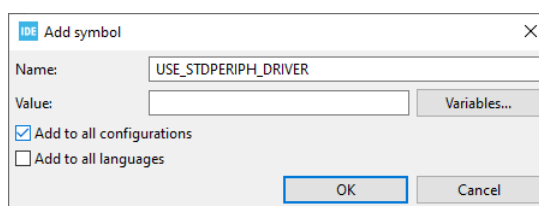
Rys. 2.13. Okno pozwalające na wybór katalogów z plikami źródłowymi

2.2. TREŚĆ ĆWICZENIA

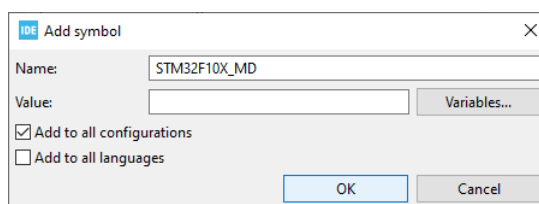
szy na korzeń projektu (rys. 2.11) i wybrać *Properties*. W oknie, które się pojawi, w lewej jego części, należy wybrać *C/C++ General* → *Paths and Symbols*. W centralnej/prawej części okna aktywuje się zakładka *Includes*, gdzie zobaczyć można będzie, przed chwilą dodane, ścieżki zawierające pliki nagłówkowe – wykorzystanie tego widoku może być alternatywą dla procesu, który wcześniej został wykorzystany. Aby wskazać kompilatorowi katalogi ze źródłami, należy przejść jednak do zakładki *Source Location* (rys. 2.12). Tutaj należy kliknąć przycisk *Add Folder...* a następnie wybrać katalogi *Libraries* oraz *Drivers* (rys. 2.13) i wcisnąć przycisk OK. Kompilator przeszuka nie tylko te konkretne katalogi, ale także wszystkie zagnieżdżone katalogi w tych katalogach.



Rys. 2.14. Okno szczegółowych ustawień projektu – symbole preprocesora



Rys. 2.15. Okno dodawania nowego symbolu preprocesora – symbol informujący kompilator o wykorzystaniu bibliotek SPL



Rys. 2.16. Okno dodawania nowego symbolu preprocesora – symbol informujący kompilator o wykorzystaniu instrukcji dla mikrokontrolera o „średniej gęstości” (*Medium Density*)

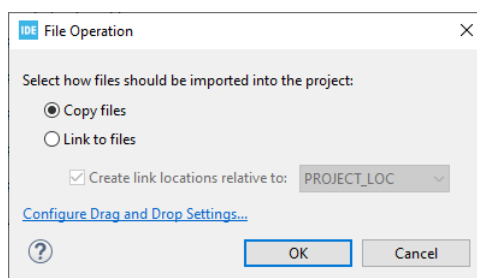
Należy także dodać dwa symbole preprocesora wykorzystywane w plikach nagłówkowych biblioteki SPL. W tym celu należy przejść do zakładki *Symbols* (rys. 2.14) i po kliknięciu przycisku *Add...* wpisać w pole *Name*: wyrażenie `USE_STDPERIPH_DRIVER`, zaznaczyć *Add to all configurations* (rys. 2.15), po czym kliknąć *OK*. Dzięki temu symbolowi biblioteka SPL zostanie aktywowana. Należy jednak dodać kolejny symbol informujący o rodzaju mikrokontrolera jaki jest wykorzystywany w niniejszym programie. W tym celu należy ponownie kliknąć *Add...* i w pole *Name*: wpisać `STM32F10X_MD`, zaznaczyć *Add to all configurations* (rys. 2.16), po czym wcisnąć *OK*. Następnie można zamknąć to okno przyciskiem *Apply and Close*.

Przy próbie kompilacji na tym etapie, kompilator zwróci uwagę na brak pliku o nazwie `stm32f10x_conf.h`, który to jest plikiem konfiguracyjnym jakie części biblioteki SPL mają faktycznie być dołączone do programu. Z katalogu wcześniej wskazanego przez prowadzącego należy więc przenieść ten plik do poddrzewa (obecnie pustego) *Inc*. Spowoduje to pojawienie się okna jak na rys. 2.17, gdzie należy wybrać kopiowanie (pierwszą opcję) i zaakceptować to przyciskiem *OK*. Drzewo projektu na tym etapie powinno wyglądać jak na rys. 2.18. W tym momencie program powinien się skompilować z powodzeniem, co w widoku konsoli powinno wyglądać podobnie jak na rys. 2.19. W przyszłości może wystąpić sytuacja, w której użytkownik rozpocznie kompilację bez zapisania uprzedniego wszystkich zmodyfikowanych plików – pojawi się wtedy okno na rys. 2.20, gdzie warto zaznaczyć aby następnym razem zmiany były automatycznie zapisywane w momencie rozpoczęcia kompilacji.

Na potrzeby dalszych rozdziałów warto jeszcze dodać możliwość wykorzystania liczb zmiennoprzecinkowych w funkcji `sprintf`. W tym celu należy ponownie wejść w opcje projektu (rys. 2.11), w drzewie z lewej strony wybrać *C/C++ Build* → *Settings*, następnie w zakładce *Tool Settings* wybrać w drzewie *MCU Settings*. Tutaj należy wybrać *Use float with printf from newlib-nano (-u _printf_float)* i wcisnąć przycisk *Apply and Close* (rys. 2.21).

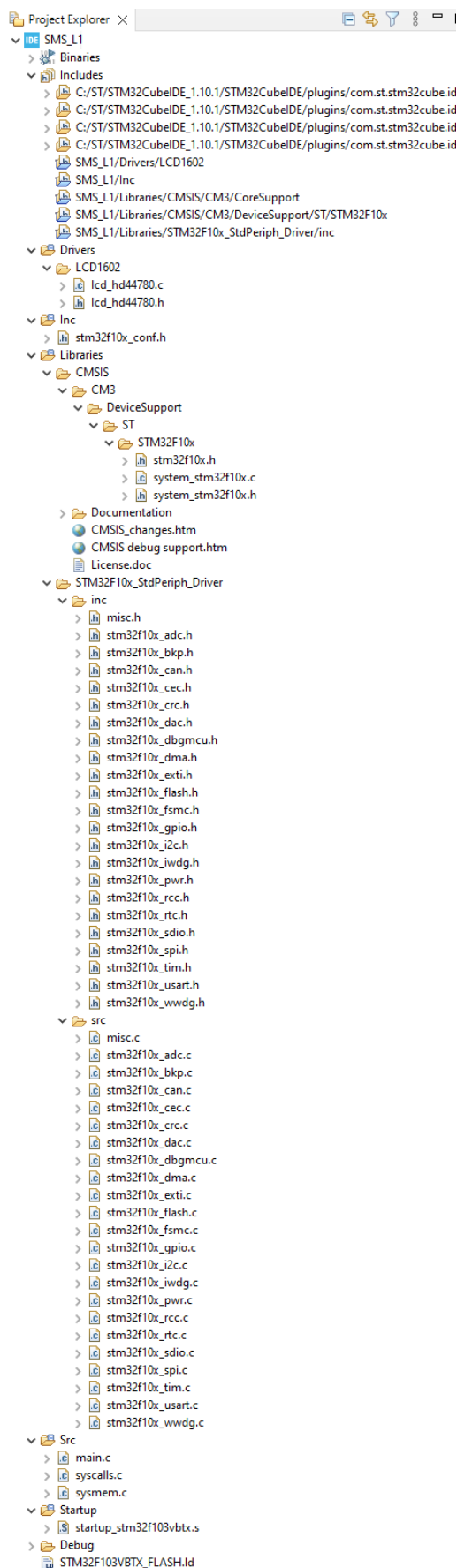
2.2.5. Wgranie programu na mikrokontroler

Taki program można już wgrać na mikrokontroler. Należy więc włączyć mikrokontroler przełącznikiem w prawym górnym rogu płytki rozwojowej, a następnie wgrać program korzystając z programatora, co wykonuje się poprzez kliknięcie przycisku zielonego kółka z białą strzałką na pasku narzędzi lub poprzez menu *Run* → *Run As* → *1 STM32 C/C++ Application*, co spowoduje pojawienie się okna widocznego na rys. 2.22. W tym oknie zdefiniowana jest nazwa konfiguracji wykorzystanej do wgrywania oprogramowania na mikrokontroler. Interesującą nas zakładką jednak będzie zakładka *Debugger* (rys. 2.24), gdzie należy zmienić programator na odpowiedni. W tym celu należy w polu *Debug pro-*

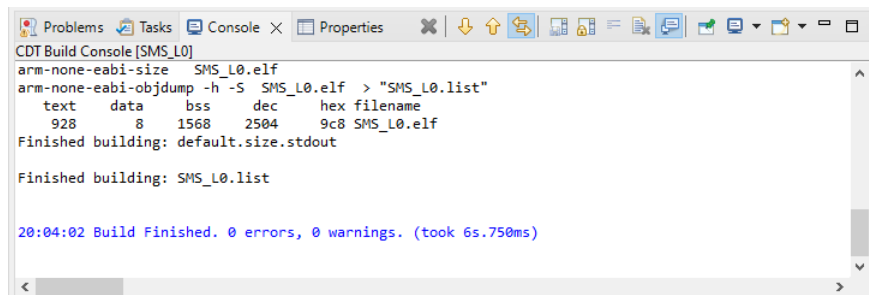


Rys. 2.17. Okno wyboru sposobu importowania plików do projektu

2.2. TREŚĆ ĆWICZENIA



Rys. 2.18. Widok drzewa projektu gotowego do implementacji pierwszego programu z wykorzystaniem biblioteki SPL



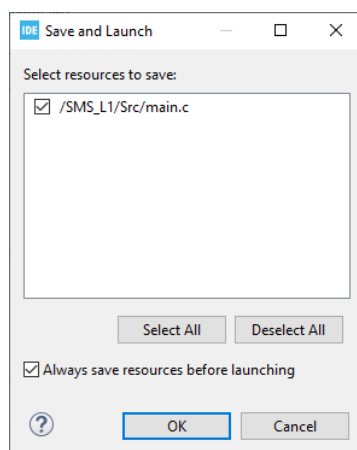
CDT Build Console [SMS_L0]

```
arm-none-eabi-size SMS_L0.elf
arm-none-eabi-objdump -h -S SMS_L0.elf > "SMS_L0.list"
text    data    bss    dec    hex filename
 928     8    1568    2504    9c8 SMS_L0.elf
Finished building: default.size.stdout

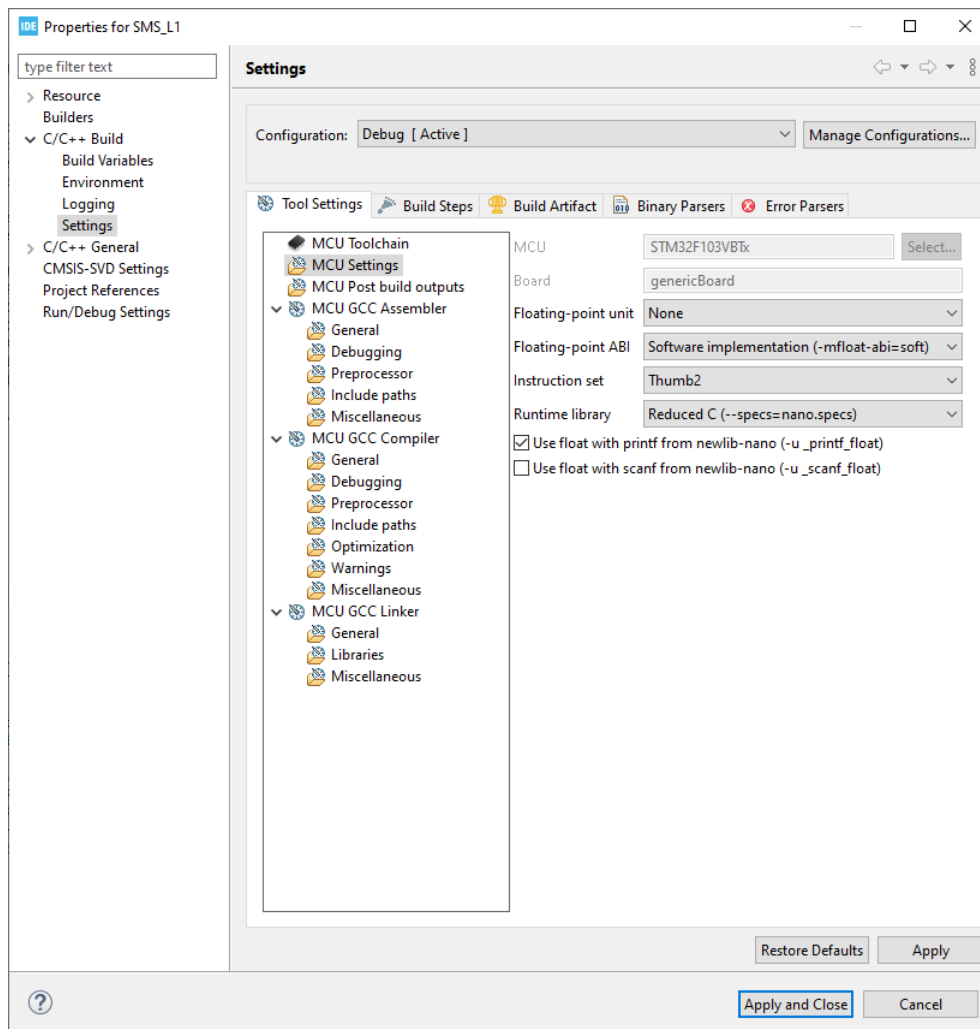
Finished building: SMS_L0.list

20:04:02 Build Finished. 0 errors, 0 warnings. (took 6s.750ms)
```

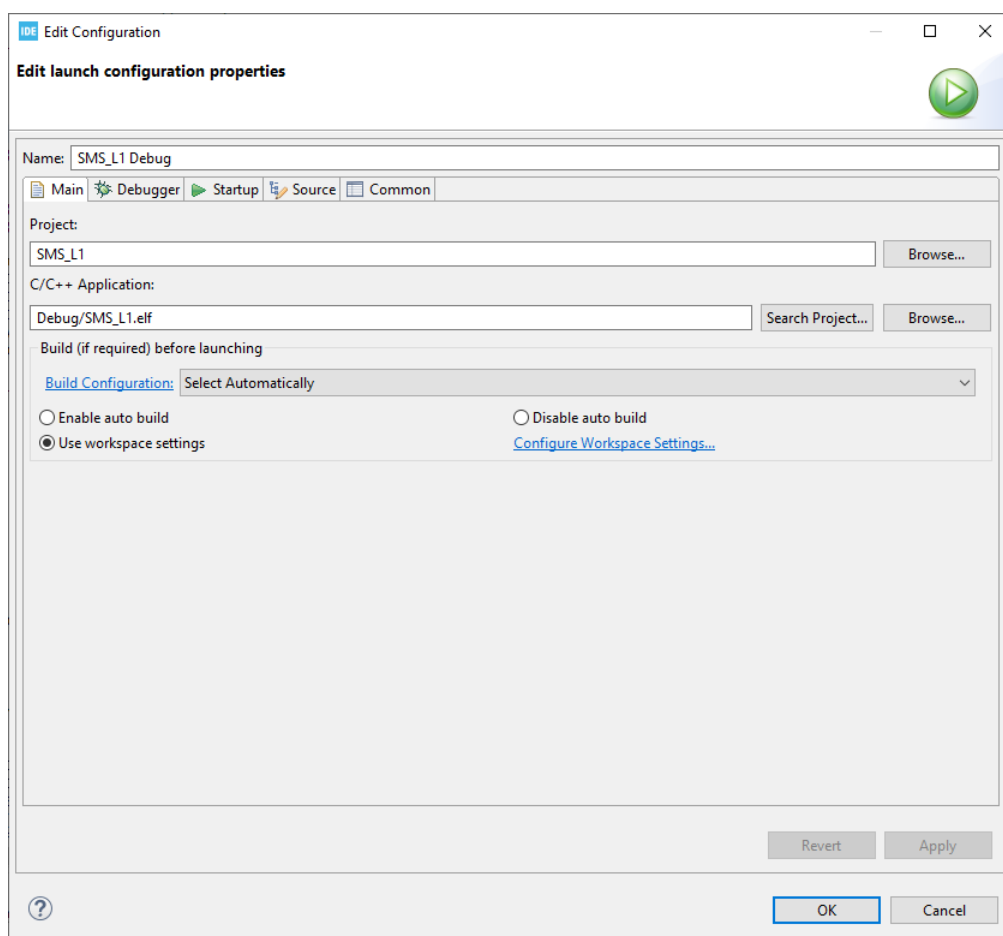
Rys. 2.19. Widok konsoli informujący o poprawnym przebiegu kompilacji



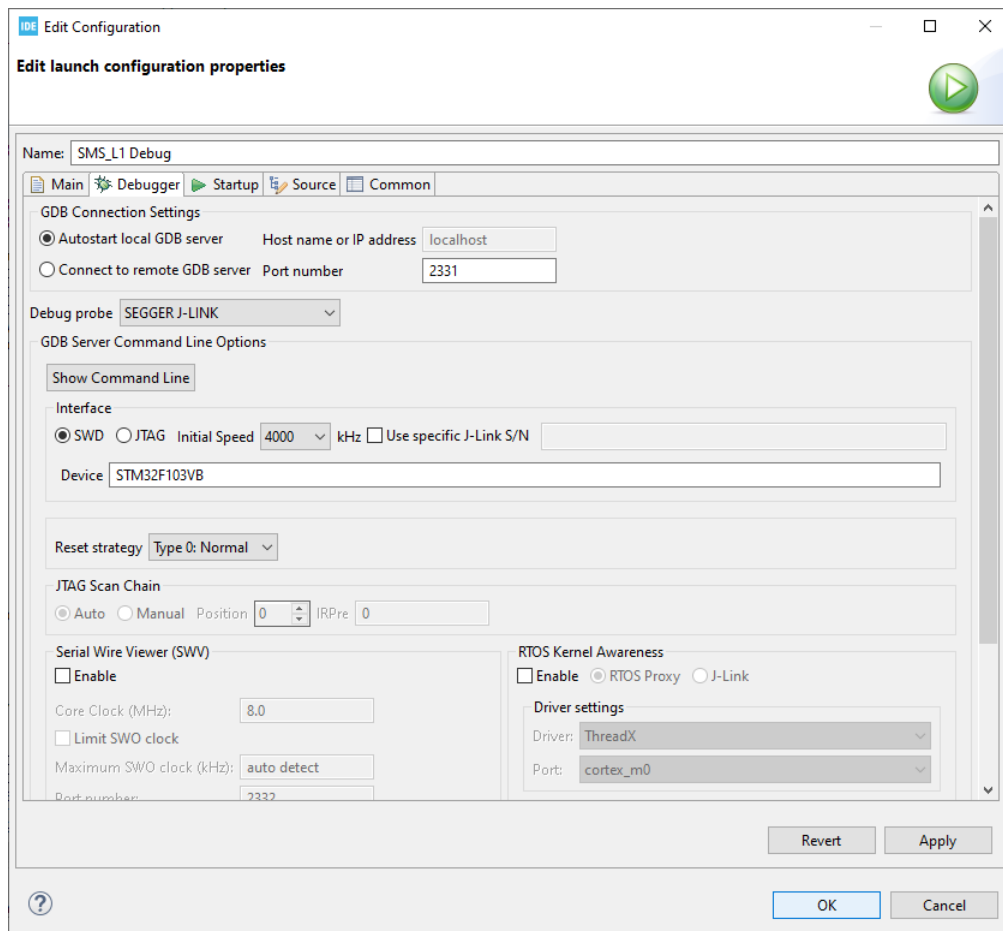
Rys. 2.20. Okno informujące o istnieniu niezapisanych zmian w momencie rozpoczęcia kompilacji



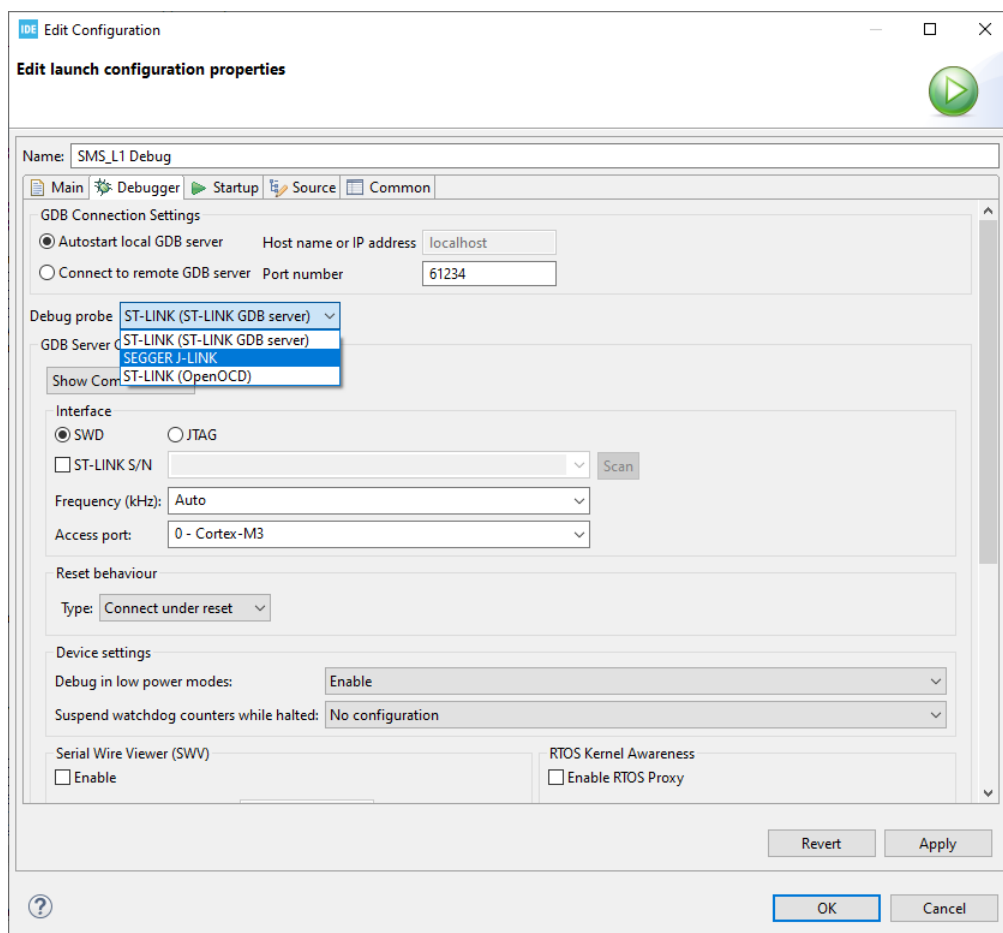
Rys. 2.21. Okno szczegółowych ustawień projektu – przydatne ustawienia kompilacji



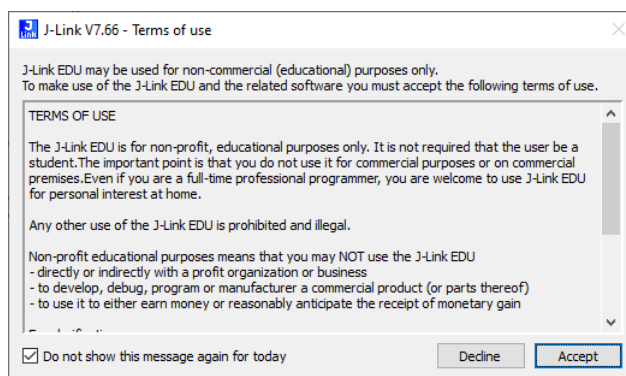
Rys. 2.22. Okno konfiguracji sposobu wgrywania programu na mikrokontroler – zakładka główna



Rys. 2.23. Okno konfiguracji sposobu wgrywania programu na mikrokontroler – zakładka konfiguracji debuggera z domyślną konfiguracją debuggera SEGGERJ-LINK



Rys. 2.24. Okno konfiguracji sposobu wgrzywania programu na mikrokontroler – zakładka konfiguracji debuggera



Rys. 2.25. Okno informujące o warunkach licencji na programator w wersji edukacyjnej

be wybrać z listy *SEGGER J-LINK*. Programator ten został wybrany, ponieważ taki właśnie sprzęt jest wykorzystany do wgrywania oprogramowania na nasz mikrokontroler. Programator ten można znaleźć obok płytki z mikrokontrolerem ZL27ARM i powinien on wyglądać jak na rys. 2.4. Po wybraniu odpowiedniego programatora powinno ukazać się okno z dodatkowymi opcjami jak na rys. 2.23 – opcje te należy jednak zostawić bez zmian i zaakceptować wszystko przyciskiem *OK*. Przy pierwszym użyciu programatora SEGGER J-LINK zostanie wyświetlony komunikat o korzystaniu z licencji edukacyjnej. Należy w tym oknie zaznaczyć, że okno to ma się nie pojawiać do końca dnia i wcisnąć przycisk akceptujący licencję (rys. 2.25). W konsoli będzie można zaobserwować komunikaty świadczące o pracy programatora, które zakończone zostaną napisem *Shutting down...*. Jest to oczekiwane zachowanie programatora, który po zakończonej procedurze wgrywania oprogramowania kończy swoją pracę – mikrokontroler jednak po wgraniu nowego programu zostanie zrestartowany przez programator i natychmiast rozpocznie jego wykonanie.

Obecny program jest najnudniejszym z możliwych, gdyż zawiera wyłącznie pustą nieskończoną pętlę. Dalsze rozdziały mają na celu opis kolejnych funkcjonalności tego mikrokontrolera, które pozwolą na jego ożywienie.

Aby rozpocząć proces debugowania wgranego programu należy wcisnąć przycisk zielonego robaka znajdujący się na pasku narzędzi. Następnie można swobodnie dodawać pułapki oraz analizować kod przy użyciu widoku *Live Expressions* zgodnie ze standardowymi schematami odrobaczania oprogramowania.

2.2.6. Konfiguracja sprzętowa mikrokontrolera

Istotne, na potrzeby tego i kolejnych ćwiczeń, elementy zestawu uruchomieniowego ZL27ARM zostały zaznaczone na rys. 2.3. Zestaw ten można uruchomić w różnych konfiguracjach. W tym ćwiczeniu oczekiwaną konfiguracją jest:

- zasilanie zestawu z portu USB,
- uruchomienie programu z wewnętrznej pamięci Flash,
- diody LED, sterowanie podświetleniem wyświetlacza LCD oraz komunikacja po USB wyłączone.

Przekłada się to na następujące ustawienia zworek na płycie ZL27ARM:

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*.

2.2.7. Konfiguracja zegarów mikrokontrolera

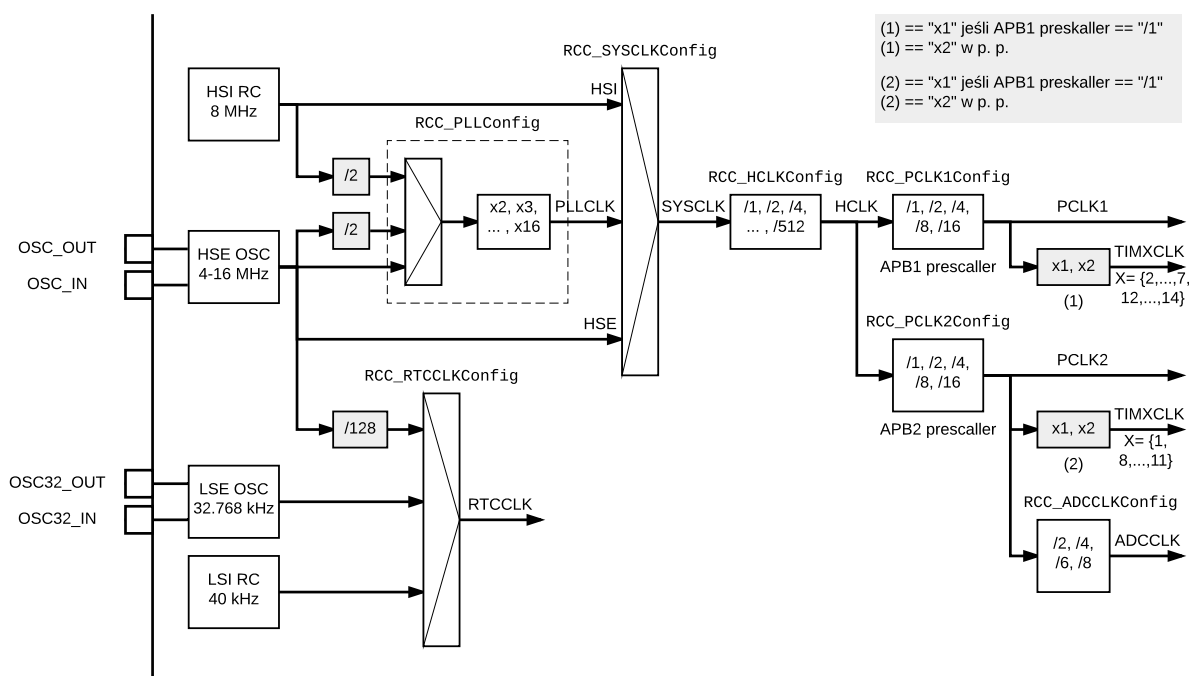
Mikrokontroler składa się z wielu (tysięcy) bramek logicznych, które pracują w trybie synchronicznym. Oznacza to, że są one taktowane zegarem i wraz z nim zmieniają wartość na wyjściu bramek. Dzięki temu unika się takich problemów jak zjawisko hazardu lub wyścigu, a więc problemów wynikających z niezerowego 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,
- *Low-Speed External* (LSE) – zewnętrzny rezonator kwarcowy 32,768 kHz,
- *High-Speed External* (HSE) – zewnętrzny rezonator kwarcowy 4 MHz-16 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 peryferali widoczny jest na rys. 2.26. Na schemacie są dodatkowo umieszczone nazwy funkcji ze stan-



Rys. 2.26. Uproszczony schemat układu taktowania procesora i peryferali – pełna wersja znajduje się w dokumencie RM0008 (Rys. 8)

dardowej biblioteki do obsługi peryferali, które pozwalają na konfigurację poszczególnych elementów i sygnałów taktujących (nazwy te są zapisane czcionką stałoszerokościową).

W tym projekcie będą wykorzystywane moduły do obsługi RCC, pamięci Flash oraz GPIO, w związku z czym należy dodać do projektu odpowiednie pliki a dokładniej wskazać, że mają zostać dołączone do projektu. Należy się upewnić, że w pliku `stm32f10x_conf.h` odkomentowane są następujące linijki:

```
1 #include "stm32f10x_flash.h"
2 #include "stm32f10x_gpio.h"
3 #include "stm32f10x_rcc.h"
```

Tak skompilowany projekt jest punktem wyjścia do konfiguracji zegarów (w oparciu o drzewo zegarów z rys. 2.26), przy której należy pamiętać o stosownej konfiguracji opóźnień odczytu z pamięci Flash. Wymaga to zastosowania prostych reguł zdefiniowanych w dokumencie 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 2s) zgodnie z poniższym kodem (`main.c`):

```
1 /*****
2  * projekt01: konfiguracja zegarow
3  *****/
4 #include "stm32f10x.h"
5
6 #define DELAY_TIME 1535000
7
8 void RCC_Config(void);
9 void GPIO_Config(void);
10 void LEDOn(void);
11 void LEDOff(void);
12 void Delay(unsigned int);
13
14 int main(void) {
15     RCC_Config();           // konfiguracja RCC
16     GPIO_Config();          // konfiguracja GPIO
17
18     while(1) {              // petla glowna programu
19         LEDOn();             // wlaczenie diody
20         Delay(DELAY_TIME);   // odczekanie 1s
21         LEDOff();            // wylaczenie diody
22         Delay(DELAY_TIME);   // odczekanie 1s
23     }
24 }
```

W powyższym kodzie należy zmodyfikować wartość stałej `DELAY_TIME` zgodnie z tabelą 2.2.7, ponieważ czas wykonania poszczególnych instrukcji, bezpośrednio zależy od częstotliwości zegara HCLK, a co za tym idzie częstotliwość HCLK wpływa pośrednio na opóźnienie generowane przez funkcję `Delay`. 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 zworę na pozycję On i ponownie włączyć mikrokontroler. Diody najprawdopodobniej rozświecą się z czasem mimo braku jakiegokolwiek 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 konfigurująca 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. 2.26).

```

1 void RCC_Config(void) {
2     ErrorStatus HSEStartUpStatus;                                // zmienna opisująca rezultat
3                                                                // uruchomienia HSE
4     // konfigurowanie sygnałów taktujących
5     RCC_DeInit();                                                // reset ustawień RCC
6     RCC_HSEConfig(RCC_HSE_ON);                                    // włącz HSE
7     HSEStartUpStatus = RCC_WaitForHSEStartUp();                 // czekaj na gotowość HSE
8     if(HSEStartUpStatus == SUCCESS) {
9         FLASH_PrefetchBufferCmd(FLASH_PrefetchBuffer_Enable); //
10        FLASH_SetLatency(FLASH_Latency_2);                     // zwłoka Flasha: 2 takty
11
12        RCC_HCLKConfig(RCC_SYSCLK_Div1);                         // HCLK=SYSCLK/1
13        RCC_PCLK2Config(RCC_HCLK_Div1);                         // PCLK2=HCLK/1
14        RCC_PCLK1Config(RCC_HCLK_Div2);                         // PCLK1=HCLK/2
15        RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9);    // PLLCLK = (HSE/1)*9
16                                                                // czyli 8MHz * 9 = 72 MHz
17        RCC_PLLCmd(ENABLE);                                       // włącz PLL
18        while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET);    // czekaj na uruchomienie PLL
19        RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);              // ustaw PLL jako źródło
20                                                                // sygnału zegarowego
21        while(RCC_GetSYSCLKSource() != 0x08);                   // czekaj aż PLL będzie
22                                                                // sygnałem zegarowym systemu
23        // konfiguracja sygnałów taktujących używanych peryferii
24        RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); // włącz taktowanie portu GPIO B
25    }
26 }

```

Nazwy funkcji są wyjątkowo długie, lecz doskonale oddają ich funkcjonalność. Znaczenie ich zostało skrótowo opisane w komentarzach. Szerszy opis można znaleźć w komentarzu nad definicją funkcji. Niestety standardowa biblioteka peryferali nie została opisana w formie dokumentacji – takową można jedynie wygenerować za pomocą narzędzia Doxygen, co jest jednak równoznaczne z czytaniem komentarzy znad 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.

Oczekiwane HCLK	DELAY_TIME
14 MHz	315000
15 MHz	400000
72 MHz	1535000

Tab. 2.1. Liczba iteracji pętli wykonywanej w ramach funkcji Delay potrzebna do realizacji opóźnienia o długości 1s przy zadanym zegarze HCLK

2.2. TREŚĆ ĆWICZENIA

Na koniec inicjalizacji warto także włączyć taktowanie peryferiali. Na razie wykorzystana zostanie wyłącznie jedna dioda znajdująca się na płycie uruchomieniowej, podłączona do pinu PB8, tj. pinu 8 portu B. Konfiguracja tego pinu znajduje się w osobnej funkcji i przebiega następująco:

```
1 void GPIO_Config(void) {
2     // konfigurowanie portow GPIO
3     GPIO_InitTypeDef GPIO_InitStructure;
4
5     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;           // pin 8
6     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;    // czestotliwosc zmiany 2MHz
7     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;    // wyjscie w trybie push-pull
8     GPIO_Init(GPIOB, &GPIO_InitStructure);             // inicjacja portu B
9 }
```

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. Nie jest to jedyna możliwa konfiguracja pinu wyjściowego, lecz jest ona najbardziej odpowiednia do sterowania diodą LED.

Warto zdefiniować przydatne funkcje służące do obsługi diody LED:

```
1 void LEDOn(void) {
2     // włączenie diody LED podłączonej do pinu 8 portu B
3     GPIO_WriteBit(GPIOB, GPIO_Pin_8, Bit_SET);
4 }
```

oraz

```
1 void LEDOff(void) {
2     // wyłączenie diody LED podłączonej do pinu 8 portu B
3     GPIO_WriteBit(GPIOB, GPIO_Pin_8, Bit_RESET);
4 }
```

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, wspomniane wcześniej, programowe opóźnienie:

```
1 void Delay(unsigned int counter){
2     // opoznienie programowe
3     while (counter--){ // sprawdzenie warunku
4         __NOP();       // No Operation
5         __NOP();       // No Operation
6     }
7 }
```

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. Jak wcześniej zostało wspomniane, wartość argumentu dająca opóźnienie równe 1 s jest zależna od zegara HCLK i na potrzeby tych ćwiczeń została wyznaczona eksperymentalnie (tabela 2.2.7). Aby więc osiągnąć opóźnienie o długości 1 s przy zegarze HCLK o częstotliwości 72 MHz należy do funkcji `Delay` przekazać wartość 1535000.

2.2.8. Odczyt wartości cyfrowej

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`):

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

Aby wciśnięcie przycisku SW0 powodowało zapalenie diody LED podłączonej do pinu 9 portu B (sąsiednia dioda w stosunku do poprzednio używanej) należy rozwinąć konfigurację zawartą w funkcji `GPIO_Config` poprzez modyfikację linijki określającej konfigurowane piny:

```
1 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:

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

Odczyt wartości cyfrowej przy użyciu tego pinu realizowany jest funkcją:

```
1 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 zwierzał podany pin do masy w momencie jego wciśnięcia. Gdy przycisk nie jest wciśnięty, zwierza 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 podciągnięcia 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 dioda LED 1, tak jak do tej pory, włączała się i wyłączała z okresem 2 s i wypełnieniem 50% (tj. dioda ma być przez 1 s i przez 1 s wyłączona) oraz aby wciśnięcie przycisku powodowało zapalenie sąsiedniej diody LED. Ćwiczenie to jednak pozostanie do wykonania dla czytelnika, gdyż ma za zadanie pokazać jakie problemy mogą wynikać z programowo realizowanego opóźnienia. Podejście to zostanie poprawione w jednym z dalszych projektów.

2.2.9. Obsługa alfanumerycznego wyświetlacza LCD

Następnym istotnym usprawnieniem omawianego programu 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`, które zawierają odpowiednio definicje i deklaracje funkcji do obsługi wyświetlacza. Znaleźć można je w katalogu *Drivers LCD1602* i został dodany do projektu wraz z jego tworzeniem.

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. W tabeli 2.2 przedstawiona jest tabela opisująca podłączenie wyświetlacza do mikrokontrolera.

Korzystanie z wyświetlacza należy rozpocząć od wywołania funkcji `LCD_Initialize`. Należy mieć świadomość, że funkcja ta zawiera konfigurację pinów potrzebnych przez wyświetlacz (zgodnie z tabelą 2.2), 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, gdyż nie jest to wykonywane w ramach inicjalizacji.

Najważniejszymi funkcjami dostępnymi w ramach tej biblioteki do obsługi wyświetlacza 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 do wyświetlacza opisuje dokładnie poszczególne komendy, które są obsługiwane przez jego sterownik. Naśladując procedurę inicjalizacji, można wywołać przykładowo komendę przesunięcia **kursora** w **prawą** stronę o jedno miejsce:

```
1 LCD_WriteCommand(HD44780_DISPLAY_CURSOR_SHIFT |
2                   HD44780_SHIFT_CURSOR |
3                   HD44780_SHIFT_RIGHT);
```

Pierwsza stała (`HD44780_DISPLAY_CURSOR_SHIFT`) określa komendę, którą przesyła się do wyświetlacza (w tym przypadku jest to *Cursor or display shift*), a następnie parametry tej komendy. W powyższym przykładzie są to: przesunięcie kursora (`HD44780_SHIFT_CURSOR`) oraz przesunięcie w prawą stronę (`HD44780_SHIFT_RIGHT`).

nazwa pinu		we/wy	opis
LCD	STM32		
RS	PC12	wy	<i>Register Select</i> , wybór rejestru: 0 – instrukcji, 1 – 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	<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
DB6	PC1	we/wy	
DB5	PC2	we/wy	
DB4	PC3	we/wy	
DB3	—	—	<i>Data Bits: b0-3</i> , trój-stanowe wejścia/wyjścia danych. W trybie z transferem 4-bitowym te cztery bity są niewykorzystywane
DB2	—	—	
DB1	—	—	
DB0	—	—	

Tab. 2.2. Opis podłączenia oraz znaczenia poszczególnych pinów wyświetlacza LCD o 16 znakach i dwóch liniach

2.2.10. Konfiguracja PWM

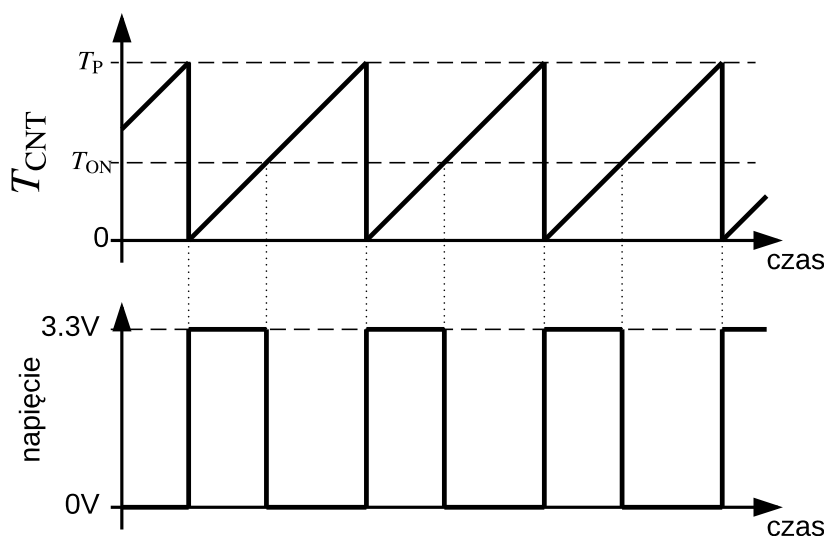
Ponieważ mikrokontrolery z rodziny STM32 są wysoce konfigurowalne, poniższy opis ograniczy się do omówienia wyłącznie konfiguracji układu timera w trybie generacji sygnału PWM. Aby wykorzystywać timery w tworzonym programie należy dodać je do konfiguracji poprzez odkomentowanie linijki

```
1 #include "stm32f10x_tim.h"
```

w pliku `stm32f10x_conf.h`.

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 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. $T_{ON} + T_{OFF} = T_P$, gdzie T_P to czas trwania pojedynczego okresu. Szerokość wspomnianego impulsu może być wyrażona w procentach jako stosunek trwania sygnału wysokiego do okresu, tj. $\frac{T_{ON}}{T_P} \cdot 100\%$. 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. 2.27 widoczne jest (w uproszczeniu) jak generowana jest fala PWM. W dalszej części poszczególne wartości będą wynosić: $T_{ON} = 1024$, $T_{OFF} = 3071$, $T_P = 4095$.

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



Rys. 2.27. Uproszczony wykres zależności między zawartością licznika T_{CNT} , a postacią wygenerowanej fali PWM

2.2. TREŚĆ ĆWICZENIA

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 jest na pinie PB8, do którego podłączony jest kanał 3 timera *TIM4* (zgodnie z tabelą 5: *Medium-density STM32F103xx pin definition*, z dokumentacji technicznej używanego mikrokontrolera – Rys. 2.28).

Ponieważ pin PB8 był wcześniej wykorzystany, to należy zaktualizować jego poprzednią konfigurację. Warto zauważyć, że skonfigurowanie jednego pinu na dwa różne sposoby nie skutkowałoby 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 PA0 jako wejścia typu GPIO_Mode_IN_FLOATING (aby móc wykorzystać podciąganie wykonane na płycie rozwojowej),
- inicjalizację pinów związanych z LED-ami, a w szczególności inicjalizacja pinu PB9, choć na tym etapie warto rozważyć konfigurację od PB9 do PB15 aby zgasić nieużywane LED-y na początku programu.

Konfiguracja pinu PB8 powinna natomiast zostać zaktualizowana do poniższej postaci:

```
1 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;           // pin 8
2 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;  // szybkość 50MHz
3 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;    // wyjście w trybie alt. push-pull
4 GPIO_Init(GPIOB, &GPIO_InitStructure);
```

Dzięki takiej konfiguracji będzie on mógł być sterowany bezpośrednio falą PWM wygenerowaną przez timer. Aby taką falę wygenerować, należy skonfigurować timer bazowy oraz co najmniej jeden jego kanał. Poniżej znajduje się konfiguracja timera bazowego (TIM4) wraz z konfiguracją jednego z jego kanałów (OC3):

```
1 void PWM_Config(void) {
2     // konfiguracja timera
3     TIM_TimeBaseInitTypeDef timerInitStructure;
4     timerInitStructure.TIM_Prescaler = 0;           // prescaler = 0
5     timerInitStructure.TIM_CounterMode = TIM_CounterMode_Up; // zliczanie w gore
6     timerInitStructure.TIM_Period = 4095;           // okres dlugosci 4095+1
7     timerInitStructure.TIM_ClockDivision = TIM_CKD_DIV1; // dzielnik czestotliwosci = 1
8     timerInitStructure.TIM_RepetitionCounter = 0;   // brak powtorzen
9     TIM_TimeBaseInit(TIM4, &timerInitStructure);   // inicjalizacja timera TIM4
10    TIM_Cmd(TIM4, ENABLE);                          // aktywacja timera TIM4
11
12    // konfiguracja kanalu timera
13    TIM_OCInitTypeDef outputChannelInit;
14    outputChannelInit.TIM_OCMode = TIM_OCMode_PWM1; // tryb PWM1
15    outputChannelInit.TIM_Pulse = 1024;              // wypelnienie 1024/4095*100% = 25%
16    outputChannelInit.TIM_OutputState = TIM_OutputState_Enable; // stan Enable
```

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⁾ /	

Rys. 2.28. Fragment noty katalogowej mikrokontrolera STM32F100, tabela 4. – rozpiska funkcji pinów

```

17 outputChannelInit.TIM_OCpolarity = TIM_OCpolarity_High; // polaryzacja Active High
18 TIM_OC3Init(TIM4, &outputChannelInit); // inicjalizacja kanału 3 timera TIM4
19 TIM_OC3PreloadConfig(TIM4, TIM_OCPreload_Enable); // konfiguracja preload register
20 }

```

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

```

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

```

Warto zwrócić uwagę na fakt, że timer ten jest podłączony do szyny *APB1* w przeciwieństwie 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:

```

1 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:

```

1 unsigned int val = 1024; // liczba 16-bitowa
2 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 peryferiów, lecz jest zazwyczaj bardziej skomplikowany i trudniejszy w czytaniu – na szczęście w tym przypadku jest to pojedynczy zapis, który jest wystarczająco intuicyjny, aby użyć go w połączeniu z biblioteką SPL. Jak widać wykorzystanie standardowej biblioteki peryferiów równoległe z pisanem do rejestrów mikrokontrolera jest możliwe i nierzadko stosowane. Dla tych, którzy wolą konsekwentnie trzymać się jednego rozwiązania: w standardowej bibliotece peryferiów znajduje się funkcja, która robi dokładnie to co powyżej (z dodatkową opcjonalną weryfikacją argumentu tej funkcji):

```

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

```

Efektom przypisania do rejestru *CCR3* timera *TIM4* wartości 1024 będzie uzyskanie słabo świecącej diody LED o numerze 1.

2.2.11. Odczyt i wykorzystanie wejścia analogowego

Aby wykonać pomiar sygnału napięciowego (tj. przetworzyć sygnał analogowy na jego cyfrową reprezentację), należy wykorzystać układ ADC. W tej sekcji omówione zostanie wykonanie konwersji poprzez programowe jej wyzwalanie.

Aby móc korzystać z ADC należy odkomentować kolejny plik nagłówkowy w pliku konfiguracyjnym `stm32f10x_conf.h`:

```

1 #include "stm32f10x_adc.h"

```

W ten sposób zostały dodane pliki do obsługi timerów i przetworników ADC, co pozwala przejść do części sprzętowej.

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 0xFF, natomiast wartość minimalna (0 V) jako 0x00. Zapis składający się z trzech znaków wynika z faktu, iż przetwornik jest 12-bitowy. Ponieważ jednak rejestry 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 konfiguracyjnym przetwornik analogowo-cyfrowy są:

- niezależne działanie przetworników ADC1 oraz ADC2,
- pomiar wyłącznie jednego kanału (nr 14) przetwornika ADC1 (tj. pomiaru napięcia na pinie PC4, gdzie podpięty jest potencjometr P1 widoczny na schemacie z rys. 2.3 – pod wyświetlaczem widoczne jest czerwone kółko, które pozwala na obracanie potencjometrem),
- start pomiaru rozpoczyna się na programowe żądanie użytkownika,
- pomiar trwać będzie możliwie krótko (tutaj 1,5 cyklu + stały czas przetwarzania 12,5 cyklu – szczegóły w RM0008, rozdział 11.6)

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

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

można przejść do implementacji opisanej konfiguracji:

```
1 void ADC_Config(void) {
2     ADC_InitTypeDef  ADC_InitStructure;
3     GPIO_InitTypeDef GPIO_InitStructure;
4
5     ADC_DeInit(ADC1); // reset ustawien ADC1
6
7     GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4; // pin 4
8     GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; // szybkość 50MHz
9     GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING; // wyjście w floating
10    GPIO_Init(GPIOC, &GPIO_InitStructure);
11
12    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent; // niezależne działanie ADC 1 i 2
13    ADC_InitStructure.ADC_ScanConvMode = DISABLE; // pomiar pojedynczego kanału
14    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE; // pomiar na zadanie
15    ADC_InitStructure.ADC_ExternalTrigConv=ADC_ExternalTrigConv_None; // programowy start
16    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; // pomiar wyrównany do prawej
17    ADC_InitStructure.ADC_NbrOfChannel = 1; // jeden kanał
18    ADC_Init(ADC1, &ADC_InitStructure); // inicjalizacja ADC1
19    ADC_RegularChannelConfig(ADC1, 14, 1, ADC_SampleTime_1Cycles5); // ADC1, kanał 14,
20    // 1.5 cyklu
21    ADC_Cmd(ADC1, ENABLE); // aktywacja ADC1
22
23    ADC_ResetCalibration(ADC1); // reset rejestru kalibracji ADC1
24    while(ADC_GetResetCalibrationStatus(ADC1)); // oczekiwanie na koniec resetu
25    ADC_StartCalibration(ADC1); // start kalibracji ADC1
26    while(ADC_GetCalibrationStatus(ADC1)); // czekaj na koniec kalibracji
27 }
```

Jak widać pin służący do pomiaru analogowej wartości napięcia został skonfigurowany jako niepodciągnięty pin wejściowy (GPIO_Mode_IN_FLOATING). Podciągnięcie takiego pinu w którymkolwiek kierunku skutkowałoby błędnymi odczytami. Warto zadać sobie jednocześnie pytanie „gdzie jest zapisana informacja, że właśnie pin PC4 będzie podłączony do kanału 14 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. 2.29. 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 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 $/6$ ($72 \text{ MHz} / 6 = 12 \text{ MHz}$) – tak też konfigurujemy ten zegar. W tym celu dodajemy do funkcji `RCC_Config` następującą liniijkę:

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

Od tego momentu przetwornik analogowo-cyfrowy będzie oczekiwał na sygnał do rozpoczęcia pomiaru, po którym będzie można odczytać przygotowaną przez niego wartość. Wykonuje się to w trzech krokach, które dla czytelności zostały opakowane w funkcję `readADC`:

```
1 unsigned int readADC(void){
2     ADC_SoftwareStartConvCmd(ADC1, ENABLE); // start pomiaru
3     while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET); // czekaj na koniec pomiaru
4     return ADC_GetConversionValue(ADC1); // odczyt pomiaru (12 bit)
5 }
```

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.

2.3. Wykonanie ćwiczenia

Student w ramach ćwiczenia ma do wykonania szereg zadań w postaci programu na płytce rozwojową ZL27ARM:

1. implementacja programu na płytce ZL27ARM, który przełącza diodę LED dołączoną do pinu PB8 z częstotliwością 0,5 Hz przy wykorzystaniu zegara HCLK o częstotliwości podanej przez prowadzącego zajęcia, z wykorzystaniem opóźnienia programowego,

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

Rys. 2.29. Fragment noty katalogowej mikrokontrolera STM32F100, tabela 4. – rozpiska funkcji pinów

2.3. WYKONANIE ĆWICZENIA

2. implementacja programu zapalającego diodę podłączoną do pinu PB9 pod wpływem wciśnięcia przycisku SW0,
3. implementacja programu wyświetlającego na wyświetlaczu LCD, w dwóch liniach, statyczny tekst „Hello World” (każde słowo w osobnej linijce).
4. implementacja programu pozwalającego na przesuwanie wyświetlanej na wyświetlaczu LCD treści w prawą stronę bez konieczności przerysowywania tekstu w pętli – przesuwanie powinno być wykonywane pod wpływem przycisku SW0,
5. implementacja programu obsługującego przetwornik ADC – pomiar z przetwornika (kanał 14 przetwornika ADC1) powinien być wyświetlany jako wartość napięcia na wyświetlaczu LCD (podpowiedź: warto użyć funkcji `sprintf`),
6. implementacja programu sterującego jasnością świecenia diody, podłączonej do pinu PB8, na podstawie pomiaru z przetwornika ADC.

Ponieważ kolejne zadania wymagają czasem nadpisania wcześniejszych funkcjonalności, należy zgłaszać postępy prowadzącemu zajęcia na bieżąco (tj. po każdym wykonanym punkcie z listy powyżej).

3. Ćwiczenie 2: obsługa prostych czujników i urządzeń wykonawczych mikroprocesorowego systemu automatyki przy wykorzystaniu systemu przerwań

3.1. Wprowadzenie

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.

3.2. Treść ćwiczenia

3.2.1. 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 przerwanie. 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:

```
1 while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET); // czekaj na koniec pomiaru
2
```

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 przerwanom.

Tabela 63 z pliku RM0008 zawiera rozpisaną 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 później 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 przerwania, 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 przerwającego 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 przerwonego 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:

```

1  unsigned char x = 10;
2  while(1){
3      if (x > 0) {
4          // (1)
5          --x;
6      } else {
7          break;
8      }
9  }
10

```

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 przerywane (`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 przerwanie, które mogą nam przeszkodzić.

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

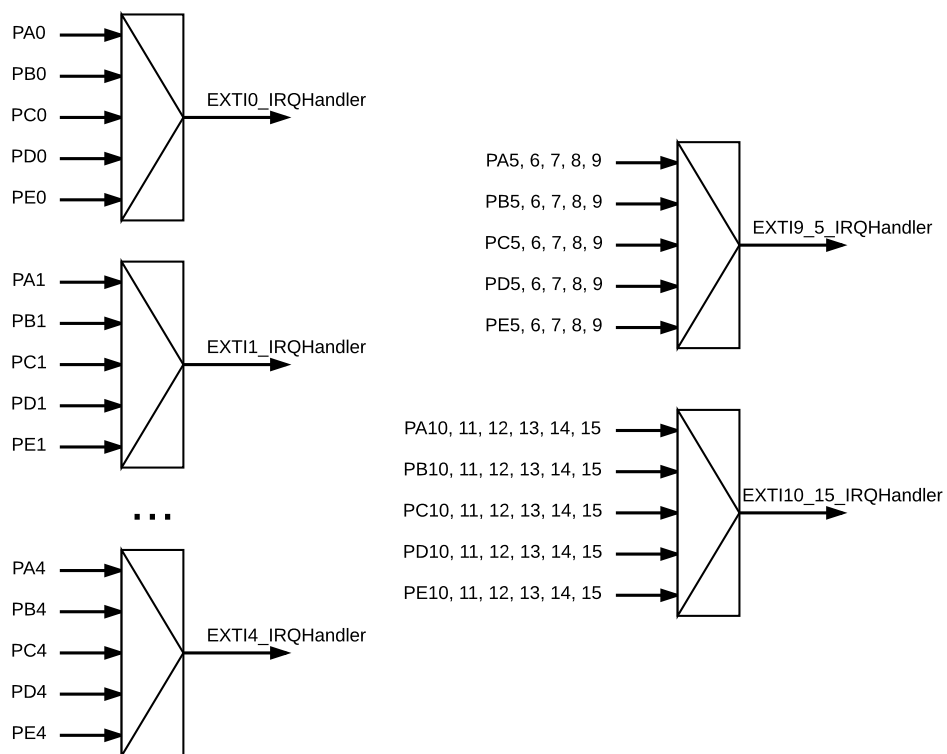
Przerwanie mogą być obsługiwane natychmiast lub zaraz po obsłudze ważniejszych przerw. W mikrokontrolerach rodziny STM32 decyzja zapada na podstawie numeru przerwania, a dokładniej poszczególnych jego bitów. Numer przerwania tworzą w rzeczywistości dwa pola, których długość można zmieniać: priorytet wyłączenia oraz pod-priorytet w obrębie priorytetu wyłączenia. 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 wyłączenia wyłącznie na podstawie wartości priorytetu (jeśli wyższy priorytet, to następuje wyłączenie), który z resztą zajmowany jest w całości przez pole priorytet wyłączenia. Pierwszy podział powoduje, że wszystkie przerwanie mają jednakowy priorytet wyłączenia, a co za tym idzie, w przypadku występowania wielu przerw 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 przerw, które muszą zostać obsłużone natychmiastowo (o mniejszym niż inne priorytecie wyłączenia), oraz takie, które mogą zostać obsłużone po innych przerwanach o tym samym priorytecie (ustawiając odpowiednio pod-priorytet).

Odpowiednie i rozważne ustawianie wartości priorytetów jest kluczowe w każdym projekcie, który wykorzystuje mechanizm przerw. W przeciwnym razie mogą nastąpić bardzo nieprzyjemne sytuacje takie jak zagłódzenie lub zakleszczenie. Jednym z przypadków, gdzie następuje zakleszczenie jest złe ustawienie priorytetu przerwania SysTick, który ma za zadanie odmierzenie 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 przerwanach SysTick, które ma niższy priorytet niż obecnie obsługiwane, to nie ma możliwości, aby wyłączyło 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.

3.2.2. Przerwanie zewnętrzne

Układ EXTI (*External interrupt / event controller*) obsługuje zewnętrzne źródła przerw – może on zgłosić przerwanie lub zdarzenie. Zdarzenia w przeciwieństwie do przerw nie muszą być obsługiwane poprzez wywołanie funkcji obsługi – mogą one bez-

3.2. TREŚĆ ĆWICZENIA



Rys. 3.1. Schemat przypisywania linii do zgłaszanego przerwania

pośrednio wywoływać pewną reakcję, np. wybudzić procesor, rozpocząć 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*, *Analog watchdog event*. Obsługa zewnętrznych źródeł przerw 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 zgłaszanego przerwania – przedstawia to rys. 3.1. 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 `EXTIO_IRQ`. Linie od 5 do 9 zgłaszają wspólne przerwanie `EXTI9_5_IRQ` (wspólne dla wszystkich portów). Analogicznie linie od 10 do 15 zgłaszają przerwanie `EXTI10_15_IRQ`.

3.2.3. Timery

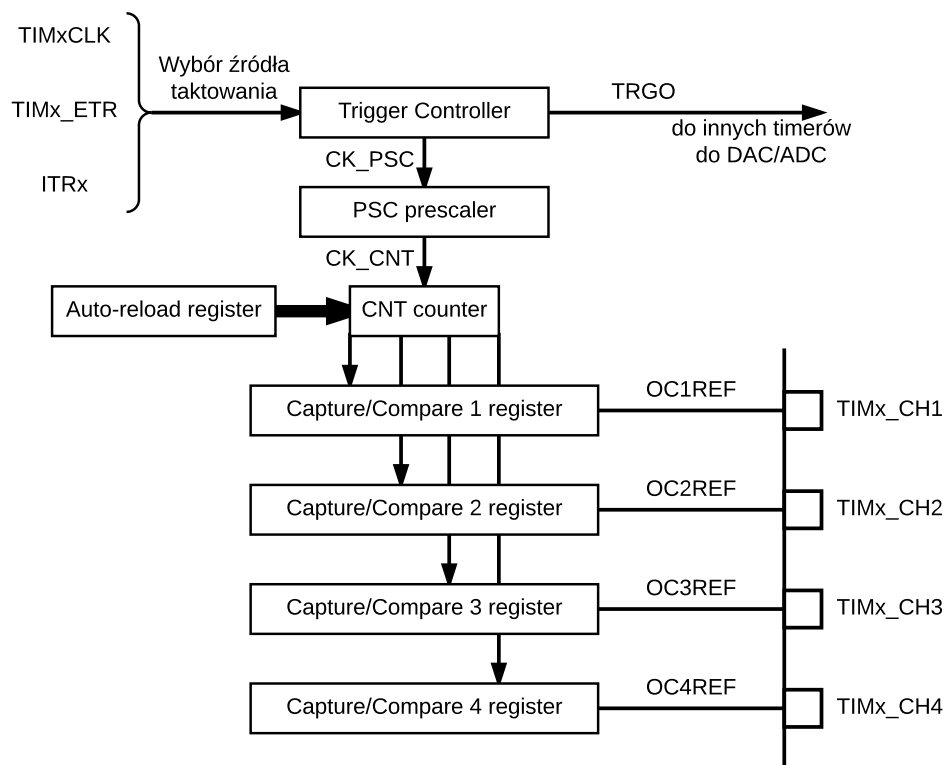
Timery, są to liczniki, których sygnałem wejściowym jest sygnał zegarowy. W mikrokontrolerach rodziny STM32 nie ma mowy o licznikach, lecz używa się właśnie pojęcia timer. W szczególnym przypadku timery służą do zliczania zboczy sygnału, który nie jest zegarowym (a więc działają jak zwykłe liczniki), 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 niezwykle skomplikowane (zazwyczaj TIM1 i TIM8, które oznaczane są jako *Advanced-control timers* i poświęcony jest im najczęściej osobny rozdział w dokumentacji).

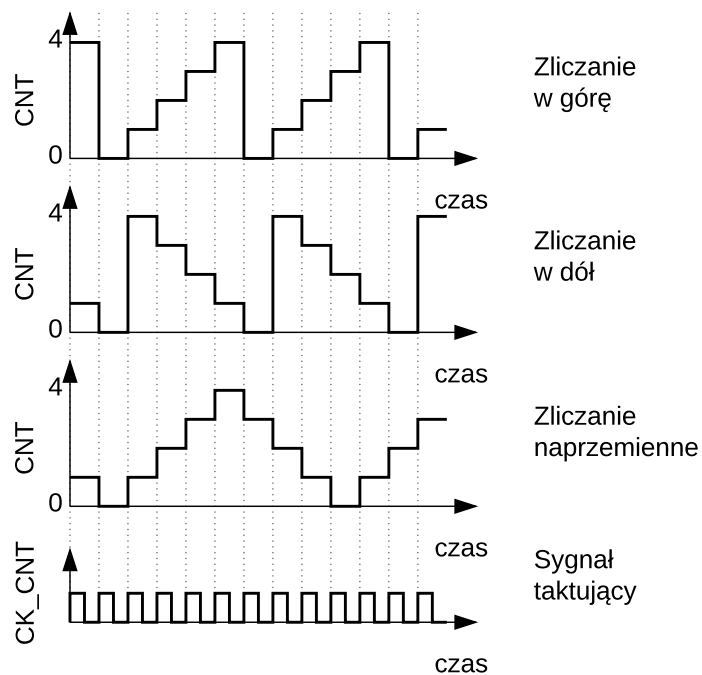
W tym ćwiczeniu w centrum uwagi będą dwa typy timerów: *Cortex System Timer* (nazywany często *SysTick*) oraz *General-purpose timer*. Pierwszy z nich należy do najprostszych w obsłudze (co wynika z niewielkich jego możliwości) timerów – zlicza on w dół takty sygnału zegarowego HCLK lub sygnału HCLK/8 (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 nada do wykorzystania w systemach czasu rzeczywistego, gdzie konieczny jest przydział kwantów czasu poszczególnym zadaniom – takie kwanty 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. 3.2. Na schemacie widać, że źródłem taktowania timera może być sygnał TIMxCLK (patrz Rys. 2.26), zewnętrzny sygnał TIMx_ETR lub wyjście innego licznika ITRx. Wybrany sygnał trafia na wejście prescalera (*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 górę, zliczania w dół, zliczania naprzemiennie w górę 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. 3.3. Przedstawiona została na nim zawartość licznika CNT w zależności od wybranego trybu zliczania, przy

3.2. TREŚĆ ĆWICZENIA



Rys. 3.2. Schemat działania timera ogólnego przeznaczenia w trybie odmierzania czasu



Rys. 3.3. Zawartość licznika CNT w zależności od wybranego trybu zliczania

założeniu, że rejestr *Auto-reload* ma wartość 4. W trybie zliczania w górę 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 górę. W przypadku gdy licznik osiągnie wartość 4 (przy zliczaniu w górę), następuje rozpoczęcie zliczania w dół. Ważną cechą tych timerów jest, że gdy do licznika wpisywane jest zero (w przypadku zliczania w górę) lub zawartość rejestru *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 *Unde-flow*.

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 takcie 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 przerwania pod kątem bardziej zaawansowanych operacji niż proste przełączanie stanu diody. Same diody należy tutaj traktować bardziej jako prymitywne narzędzie do diagnostyki (na zasadzie „jeśli miga 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.

3.2.4. 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.

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

3.2. TREŚĆ ĆWICZENIA

```
1  #include "stm32f10x.h" // definicja typu uint16_t i stałych GPIO_Pin_X
2
3  #define LED1 GPIO_Pin_8
4  #define LED2 GPIO_Pin_9
5  #define LED3 GPIO_Pin_10
6  #define LED4 GPIO_Pin_11
7  #define LED5 GPIO_Pin_12
8  #define LED6 GPIO_Pin_13
9  #define LED7 GPIO_Pin_14
10 #define LED8 GPIO_Pin_15
11 #define LEDALL (LED1|LED2|LED3|LED4|LED5|LED6|LED7|LED8)
12 enum LED_ACTION { LED_ON, LED_OFF, LED_TOGGLE };
13
14 void LED(uint16_t led, enum LED_ACTION act);
15
```

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

```
1  void LED(uint16_t led, enum LED_ACTION act) {
2      switch(act){
3          case LED_ON: GPIO_SetBits(GPIOB, led); break;
4          case LED_OFF: GPIO_ResetBits(GPIOB, led); break;
5          case LED_TOGGLE: GPIO_WriteBit(GPIOB, led,
6              (GPIO_ReadOutputDataBit(GPIOB, led) == Bit_SET?Bit_RESET:Bit_SET));
7      }
8  }
9
```

3.2.5. 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:

```
1  SysTick_Config(Ticks);
2
```

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:

```
1  SysTick_CLKSourceConfig(SysTick_CLKSource_X);
2
```

gdzie `SysTick_CLKSource_X` definiuje sygnał wejściowy dla timera SysTick. Do wyboru są dwie opcje: sygnał HCLK – `SysTick_CLKSource_HCLK` lub `SysTick_CLKSource_HCLK_Div8`, czyli sygnał HCLK/8. Wybór odpowiedniego sygnału taktującego jest bardzo ważny, gdyż licznik timera SysTick jest 24-bitowy, więc przy sygnale wejściowym HCLK przepełniać 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 $HCLK=72\text{ MHz}$, od około $0,54\text{ Hz}$ (okres około $1,86\text{ s}$) do 9 MHz (okres około $111,11\text{ ns}$). 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:

```
1 SysTick_Config(9000); // (72MHz/8) / 9000 = 1KHz (1/1KHz = 1ms)
2 SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8);
3
```

Obsługa przerwania generowanego przez SysTick sprowadza się do implementacji funkcji:

```
1 void SysTick_Handler(void);
2
```

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:**

```
1 void DelayTick(void){
2     // dekrementacja licznika (o ile jest wiekszy od 0)
3 }
4
```

```
1 void Delay(unsigned int ms){
2     // zmiana wartosci licznika
3     // testowanie czy jest on wiekszy od 0
4 }
5
```

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 ustawienie jego priorytetu na dość wysokim poziomie (tj. należy obniżyć jego wartość). Rozsądną z punktu widzenia tego ćwiczenia jest wartość 0. **Zmianę priorytetu przerwania generowanego przez timer SysTick** realizuje się przy użyciu

```
1 NVIC_SetPriority(SysTick_IRQn, 0);
2
```

uprzednio konfigurując 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.

3.2.6. Ustawienie podziału priorytetów przerw

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

```
1 NVIC_PriorityGroupConfig(NVIC_PriorityGroup_X);  
2
```

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

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

3.2.7. 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 górę,
- włączona obsługa przerwania zgłaszanego 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 przepełniał 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 zgłaszanego 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 peryferali:

```

1 TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
2 TIM_OCInitTypeDef TIM_OCInitStructure;
3
4 TIM_TimeBaseStructure.TIM_Prescaler = 7200-1; // 72MHz/7200=10kHz
5 TIM_TimeBaseStructure.TIM_Period = 10000; // 10kHz/10000=1Hz (1s)
6 TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; // zliczanie w gore
7 TIM_TimeBaseStructure.TIM_RepetitionCounter = 0; // brak powtorzen
8 TIM_TimeBaseInit(TIM4, &TIM_TimeBaseStructure); // inicjalizacja TIM4
9 TIM_ITConfig ( TIM4, TIM_IT_CC2 | TIM_IT_Update, ENABLE ); // wlaczenie przerwan
10 TIM_Cmd(TIM4, ENABLE); // aktywacja timera TIM4
11
12 // konfiguracja kanalu 2 timera
13 TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_Timing; // brak zmian OCxREF
14 TIM_OCInitStructure.TIM_Pulse = 2000; // wartosc do porownania
15 TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; // wlaczenie kanalu
16 TIM_OC2Init(TIM4, &TIM_OCInitStructure); // inicjalizacja CC2
17

```

Konfiguracja przerwania związanego z timerem następuje poprzez wykorzystanie modułu NVIC:

```

1 NVIC_InitTypeDef NVIC_InitStructure;
2
3 NVIC_ClearPendingIRQ(TIM4_IRQn); // wyczyszczenie bitu przerwania
4 NVIC_EnableIRQ(TIM4_IRQn); // wlaczenie obsługi przerwania
5 NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn; // nazwa przerwania
6 NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2; // priorytet wywlaszczania
7 NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1; // podpriorytet
8 NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; // wlaczenie
9 NVIC_Init(&NVIC_InitStructure); // inicjalizacja struktury
10

```

Po wprowadzeniu powyższych konfiguracji można **napisać funkcję obsługującą przerwanie**:

```

1 void TIM4_IRQHandler(void){
2     if(TIM_GetITStatus(TIM4,TIM_IT_CC2) != RESET){
3         LED(LED2,LED_TOGGLE);
4         TIM_ClearITPendingBit(TIM4, TIM_IT_CC2);
5     } else if(TIM_GetITStatus(TIM4,TIM_IT_Update) != RESET){
6         LED(LED3,LED_TOGGLE);
7         TIM_ClearITPendingBit(TIM4, TIM_IT_Update);
8     }
9 }
10

```

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 rejestru 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 0,2s. Na koniec każdego programu do obsługi przerwań **koniecznie trzeba wyczyścić bit świadczący o oczekiwaniu na obsługę tego przerwania**. Wykonuje się to (w przypadku timerów) przy użyciu funkcji TIM_ClearITPendingBit. W przypadku niewykonania tej operacji przerwanie to będzie nieustannie fałszywie zgłaszane jako nieobsłużone.

3.2.8. 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. 3.4 widoczny

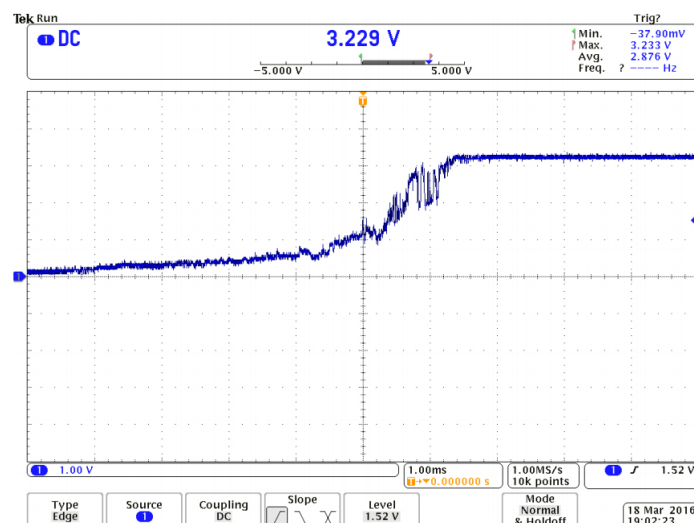
3.2. TREŚĆ ĆWICZENIA

jest przykład pomiaru wartości napięcia na przełączniku w momencie jego wciśnięcia. 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:

```
1 while(1){
2     // ...
3     stan_przelacznika = GPIO_ReadInputDataBit(GPIOx, GPIO_Pin_y);
4     // ...
5 }
6
```

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 wciśniętym, 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 oczekiwanie na zakończenie działania. Jest to niemal konieczne w przypadku interak-



Rys. 3.4. Oscylogram zjawiska drgań styków (wciśnięcie przełącznika)

cji 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. 3.4 stan przełącznika jest widziany często właśnie z taką dokładnością – każda jego zmiana jest rejestrowana i zgłaszana za pomocą mechanizmu przerwań. Zamiast więc uzyskać jedną pewną informację o wciśnięciu 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 odczekaj 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 satysfakcjonująca.

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

```
1  GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0); // PA0 -> EXTI0_IRQn
2  EXTI_InitStructure.EXTI_Line = EXTI_Line0; // linia : 0
3  EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt; // tryb : przerwanie
4  EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling; // zbocze: opadające
5  EXTI_InitStructure.EXTI_LineCmd = ENABLE; // aktywowanie konfigur.
6  EXTI_Init(&EXTI_InitStructure); // inicjalizacja
7
8  NVIC_ClearPendingIRQ(EXTIO_IRQn); // czyszcz. bitu przerw.
9  NVIC_EnableIRQ(EXTIO_IRQn); // włączenie przerwania
10 NVIC_InitStructure.NVIC_IRQChannel = EXTIO_IRQn; // przerwanie EXTIO_IRQn
11 NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; // prior. wywłaszczania
12 NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0; // podpriorytet
13 NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; // aktywowanie konfigur.
14 NVIC_Init(&NVIC_InitStructure); // inicjalizacja
15
```

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):

```
1  TIM_TimeBaseStructure.TIM_Prescaler = 7200-1; // 72MHz/7200=10kHz
2  TIM_TimeBaseStructure.TIM_Period = 350; // 10kHz/350~=29Hz (35ms)
3  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; // zliczanie w gore
4  TIM_TimeBaseStructure.TIM_RepetitionCounter = 0; // brak powtorzen
5  TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure); // inicjalizacja TIM3
6  TIM_ITConfig ( TIM3, TIM_IT_Update, DISABLE ); // wyłączenie przerwan
7  TIM_Cmd(TIM3, DISABLE); // wyłączenie timera
8
9  NVIC_ClearPendingIRQ(TIM3_IRQn); // czyszcz. bitu przerw.
10 NVIC_EnableIRQ(TIM3_IRQn); // włączenie przerwania
```


3.2. TREŚĆ ĆWICZENIA

```
11  NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;           // nazwa przerwania
12  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1; // prior. wyłaszczania
13  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;        // podpriorytet
14  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;          // aktywowanie konfiguracji
15  NVIC_Init(&NVIC_InitStructure);                          // inicjalizacja
16
```

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 zbocza opadającego na linii podłączonej do przełącznika. Z tego powodu należy następująco **zaimplementować obsługę tego przerwania**:

```
1  void EXTI0_IRQHandler(void){
2      if(EXTI_GetITStatus(EXTI_Line0) != RESET){           // sprawdzenie przyczyny
3          EXTI_ClearITPendingBit(EXTI_Line0);              // wyczyszczenie bitu przerwania
4
5          TIM_SetCounter(TIM3, 0);                          // reset licznika timera
6          TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE);       // aktywacja przerwania
7          TIM_Cmd(TIM3, ENABLE);                           // aktywacja timera TIM3
8      }
9  }
10
```

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:

```
1  void TIM3_IRQHandler(void){
2      if(TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET){ // sprawdzenie przyczyny
3          TIM_ClearITPendingBit(TIM3, TIM_IT_Update);    // wyczyszczenie bitu przerw.
4          if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0) == Bit_RESET) // jeśli wciśnięty
5              LED(LED5, LED_TOGGL);                     // zrob cos (przełącz LED5)
6          TIM_ITConfig(TIM3, TIM_IT_Update, DISABLE);    // deaktywacja przerwania
7          TIM_Cmd(TIM3, DISABLE);                        // deaktywacja timera TIM3
8      }
9  }
10
```

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.

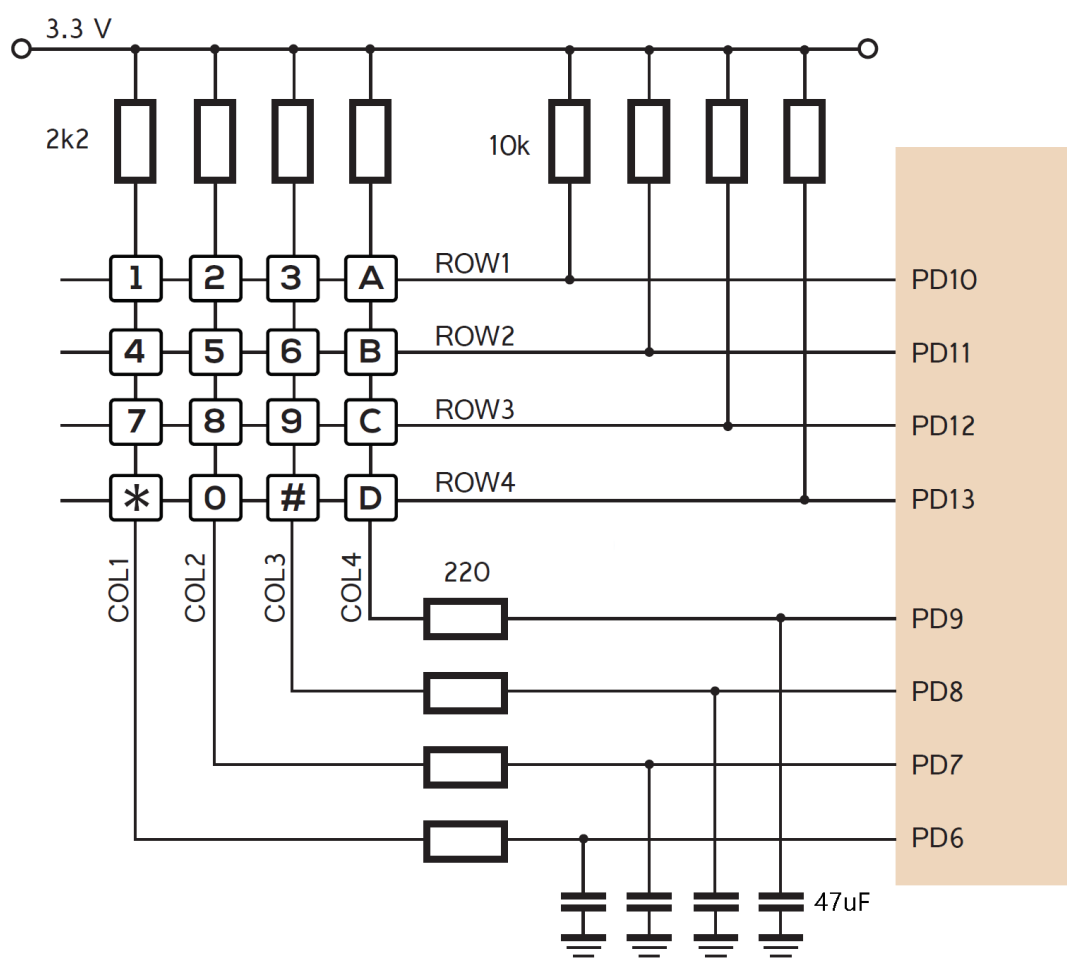
3.2.9. Obsługa klawiatury numerycznej

Do zestawu ZL27ARM można bardzo łatwo podpiąć klawiaturę o 4 kolumnach i 4 wierszach (Rys. 3.5). 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. 3.6. Na schemacie widoczne są rezystory 10 kΩ podciągające linie ROW1, ROW2, ROW3 i ROW4 do napięcia zasilania, rezystory 2,2 kΩ podciągające linie COL1, COL2, COL3, COL4 do napięcia zasilania oraz pary rezystor-kondensator realizujące filtr dolnoprzepustowy. W każdej z par rezystor ma wartość 220 kΩ, a kondensator 47 μF.

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



Rys. 3.5. Klawiatura 4×4



Rys. 3.6. Schemat podłączenia klawiatury 4×4 (źródło: *Systemy Mikroprocesorowe w Sterowaniu. Część I: ARM Cortex-M3*)

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 V). Piny od 6 do 9 są również podciągnięte do zasilania, co powoduje, że domyślnie występuje na nich właśnie 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 jeśli któryś z klawiszy w pierwszym rzędzie jest wciśnięty, 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 wciśnięte – możliwości tej nie daje jednak już sama budowa klawiatury. Odpowiednia kombinacja klawiszy może spowodować błędne wykrycie klawiszy niewciśniętych.

Wybrane do tego zadania tryby linii portu D są kluczowe dla bezpieczeństwa mikrokontrolera. Dla takiej konfiguracji wciśnięcie kilku klawiszy klawiatury jednocześnie nie spowoduje zwarcia – nie można wywołać sytuacji kiedy masa 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. Wciśnięcie w tym momencie klawiszy z rzędów pierwszego i któregośkolwiek innego spowodowałoby zwarcie masy z zasilaniem, a zarazem prawdopodobnie uszkodzenie płyty uruchomieniowej.

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

```

1 char KB2char(void){
2     unsigned int GPIO_Pin_row, GPIO_Pin_col, i, j;
3     const unsigned char KBkody[16] = {'1','2','3','A',\
4     '4','5','6','B',\
5     '7','8','9','C',\
6     '*', '0', '#', 'D'};
7     GPIO_SetBits(GPIOD, GPIO_Pin_10|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13);
8     GPIO_Pin_row = GPIO_Pin_10;
9     for(i=0;i<4;++i){
10        GPIO_ResetBits(GPIOD, GPIO_Pin_row);
11        Delay(5);
12        GPIO_Pin_col = GPIO_Pin_6;
13        for(j=0;j<4;++j){
14            if(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_col) == 0){
15                GPIO_ResetBits(GPIOD, GPIO_Pin_10|GPIO_Pin_11|
16                GPIO_Pin_12|GPIO_Pin_13);
17                return KBkody[4*i+j];
18            }
19            GPIO_Pin_col = GPIO_Pin_col << 1;
20        }
21        GPIO_SetBits(GPIOD, GPIO_Pin_row);
22        GPIO_Pin_row = GPIO_Pin_row << 1;
23    }
24    GPIO_ResetBits(GPIOD, GPIO_Pin_10|GPIO_Pin_11|GPIO_Pin_12|GPIO_Pin_13);
25    return 0;
26 }
27

```

Powyższy kod może być wykorzystany w pętli głównej programu do odpytywania (tj. nieustannego skanowania) klawiatury w celu ustalenia, które klawisze są wciśnięte (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 zbocza opadającego powoduje wyzwolenie wspólnego przerwania EXTI_Line9_5. Oznacza to, że bez względu na to, który klawisz zostanie wciśnięty, 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 wciśnięciem klawisza na klawiaturze. Nie należy implementować tutaj mechanizmu niwelacji drgań styków.

3.2.10. Okresowe wykonywanie pomiarów

Do tej pory wykonywanie analogowo-cyfrowego przetwarzania 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 rozpoczynało się ze stałą częstotliwością. Dodatkowo w trakcie wykonywania przetwarzania warto zwolnić procesor, aby nie oczekiwać 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:

```
1 TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;  
2 TIM_OCInitTypeDef TIM_OCInitStructure;  
3  
4 TIM_TimeBaseStructure.TIM_Prescaler = 7200-1;  
5 TIM_TimeBaseStructure.TIM_Period = 5000; // 2Hz -> 0.5s  
6 TIM_TimeBaseStructure.TIM_RepetitionCounter = 0;  
7 TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;  
8 TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);  
9  
10 // konfiguracja kanału timera  
11 TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;  
12 TIM_OCInitStructure.TIM_Pulse = 1; // minimalne przesunięcie  
13 TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;  
14 TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;  
15 TIM_OC2Init(TIM2, &TIM_OCInitStructure);  
16 TIM_ITConfig ( TIM2, TIM_IT_CC2 , ENABLE );  
17 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:

```
1 void ADC_Config(void) {  
2     ADC_InitTypeDef ADC_InitStructure;  
3  
4     ADC_DeInit(ADC1); // reset ustawień ADC1  
5     ADC_InitStructure.ADC_Mode = ADC_Mode_Independent; // niezależne działanie ADC 1 i 2  
6     ADC_InitStructure.ADC_ScanConvMode = DISABLE; // pomiar pojedynczego kanału  
7     ADC_InitStructure.ADC_ContinuousConvMode = DISABLE; // pomiar automatyczny  
8     ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T2_CC2; // T2CC2->ADC
```

3.2. TREŚĆ ĆWICZENIA

```
9  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; // pomiar wyrownany do prawej
10 ADC_InitStructure.ADC_NbrOfChannel = 1; // jeden kanal
11 ADC_Init(ADC1, &ADC_InitStructure); // inicjalizacja ADC1
12
13 ADC-RegularChannelConfig(ADC1, ADC_Channel_16, 1, ADC_SampleTime_41Cycles5); // konf.
14 ADC_ExternalTrigConvCmd(ADC1, ENABLE);
15 ADC_ITConfig(ADC1, ADC_IT_EOC, ENABLE);
16
17 ADC_Cmd(ADC1, ENABLE); // aktywacja ADC1
18
19 ADC_ResetCalibration(ADC1); // reset rejestru kalibracji ADC1
20 while(ADC_GetResetCalibrationStatus(ADC1)); // oczekiwanie na koniec resetu
21 ADC_StartCalibration(ADC1); // start kalibracji ADC1
22 while(ADC_GetCalibrationStatus(ADC1)); // czekaj na koniec kalibracji
23
24 ADC_TempSensorVrefintCmd(ENABLE); // włączenie czujnika temperatury
25 }
```

Do głównych różnic należy **wyбір 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 wywołało uruchomienie funkcji obsługi przerwania, **wyczyścić** bity oczekujących przerw i **pobrać** wartość przetworzoną przez przetwornik:

```
1 void ADC1_2_IRQHandler(void){
2     if(ADC_GetITStatus(ADC1, ADC_IT_EOC) != RESET){
3         ADC_ClearITPendingBit(ADC1, ADC_IT_EOC);
4         sprintf((char*)bufor, "%2d*C", ((V25-ADC_GetConversionValue(ADC1))/Avg_Slope+25));
5     }
6 }
```

Wartości V25 i Avg_Slope są zdefiniowane jako:

```
1 const uint16_t V25 = 1750; // gdy V25=1.41V dla napięcia odniesienia 3.3V
2 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_IRQn)**– 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.

3.2.11. Dodatkowe informacje

Warto pamiętać o kolejności konfiguracji kolejnych peryferali:

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 funkcji `NVIC_EnableIRQ`,
 6. implementacja obsługi przerwania – wypełnienie funkcji z pliku `stm32f10x_it.c`.

3.3. Wykonanie ćwiczenia

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 drgań 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 wyświetlanej 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 klawisza).

W przeciwieństwie do ćwiczenia pierwszego, tym razem wszystkie powyższe punkty można (a nawet należy) zaprezentować jednocześnie. Poprzez poprawne wykorzystanie mechanizmu przerwań dodawanie kolejnych funkcjonalności nie powinno wpływać w widoczny sposób na działanie poprzednich.

4. Ćwiczenie 3: Obsługa złożonych czujników i urządzeń wykonawczych mikroprocesorowego systemu automatyki

4.1. Wprowadzenie

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 znikomym podzbiór standardów komunikacyjnych, lecz pozwala on uświadomić, że nawet wyjątkowo proste rozwiązania mogą być i są skutecznie implementowane w przemyśle.

4.2. Treść ćwiczenia

4.2.1. 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. Załóżmy, 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 160 Ω .

Zastosowanie prądu do reprezentacji pomiaru posiada wiele pożytecznych (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.

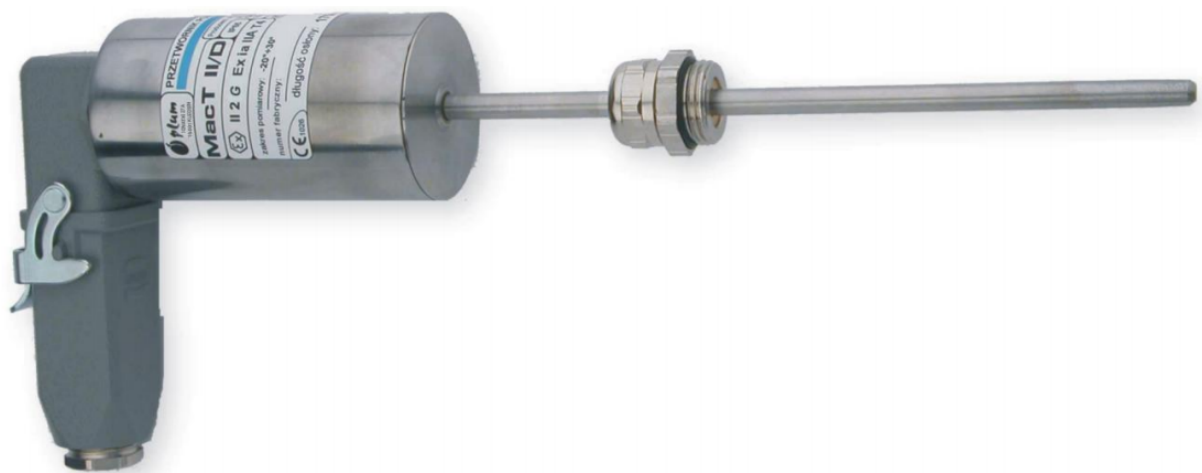
W związku z powyższym przemysłowe czujniki mogą być podłączone bardzo długimi kablami do urządzeń odczytujących pomiary. Pomiar z wykorzystaniem napięcia w takiej sytuacji byłby obciążony znaczącym błędem.

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

4.2.2. 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. 4.1). 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 iskrobezpiecznie i może pracować w strefach zagrożenia wybuchem 1 i 2.

Powyższy przetwornik należy zasilć 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 szereg 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 zasilć 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. 4.2. Należy zwrócić szczególną uwagę na podłączenie do mikrokontrolera – zamiana masy płytki z pinem wejściowym może spowodować uszkodzenie płytki rozwojowej. Dla ułatwienia podłączeń



Rys. 4.1. Cyfrowy przetwornik temperatury typu MacT II firmy Plum.

została zrealizowana płytki 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 (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:

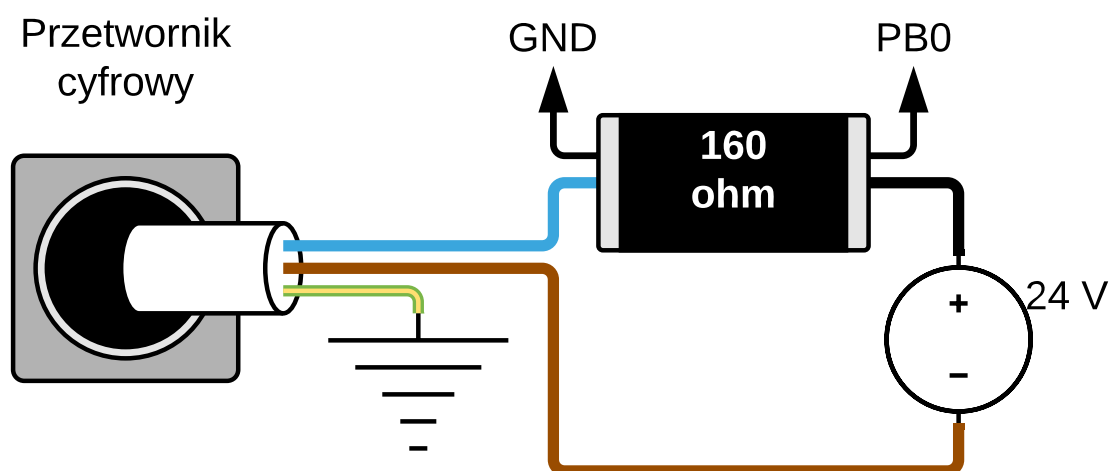
$$U(t) = \left(\frac{t + 30}{90}16\text{ mA} + 4\text{ mA} \right) 160\Omega = \frac{t + 30}{90}2,56\text{ V} + 0,64\text{ V}$$

Warto jednak zauważyć, 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 rozwiązaniu 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,



Rys. 4.2. Schemat podłączenia przetwornika do zasilacza w celu pomiaru temperatury.

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 uprościć tę funkcję, tak aby była wygodniejsza w implementacji i nie wymagała tak dużej liczby obliczeń.

4.2.3. Transmisja szeregową

Transmisja szeregową (w przeciwieństwie 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 płycie 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 szeregową 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 wysłał.

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 wyznaczone 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 szeregową 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, dzielonego 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 ze stron. 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 tyle właśnie zajmuje jeden taki znak.

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 szeregową 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. 4.3 (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.

4.2.4. Implementacja na ZL27ARM

Implementację transmisji szeregową 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 szeregową jest funkcją alternatywną tych pinów

Test parzystości				Liczba bitów stopu				Długość danych			
brak	0	11110010	1	1	0	11110010	1	8	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

Rys. 4.3. Możliwości konfiguracji wiadomości w transmisji szeregową. Oznaczenia kolorystyczne: pomarańczowy – bit startu, zielony – dane, szary – bit parzystości, niebieski – bity stopu.

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

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

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

```
1  GPIO_InitTypeDef GPIO_InitStructure;
2
3  // Pin nadawczy należy skonfigurować jako "alternative function, push-pull"
4  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
5  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
6  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
7  GPIO_Init(GPIOA, &GPIO_InitStructure);
8
9  // Pin odbiorczy należy skonfigurować jako wejście "pływakowe"
10 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
11 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
12 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
13 GPIO_Init(GPIOA, &GPIO_InitStructure);
```

Dalej następuje konfiguracja samej transmisji szeregowej:

```
1  USART_InitTypeDef USART_InitStructure;
2  USART_InitStructure.USART_BaudRate = 19200;
3  USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
4  USART_InitStructure.USART_WordLength = USART_WordLength_9b;
5  USART_InitStructure.USART_Parity = USART_Parity_Even;
6  USART_InitStructure.USART_StopBits = USART_StopBits_1;
7  USART_InitStructure.USART_Mode = USART_Mode_Tx | USART_Mode_Rx;
8
9  USART_Init(USART1, &USART_InitStructure);
10 USART_ITConfig(USART1, USART_IT_RXNE, DISABLE);
11 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 peryferiali).

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:

```
1  NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn;
2  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
3  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
4  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
5  NVIC_Init(&NVIC_InitStructure);
```

Dopiero po tym etapie należy włączyć USART:

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

4.2.5. Transmisja szeregową – 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 $t_{1,5}$. 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 $t_{1,5}$ oraz $t_{3,5}$ (omówione zostaną za chwilę). Z poziomu urządzenia typu *master* (gdyż taki będzie nas interesował) nie można wysłać 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 $t_{1,5}$ oraz $t_{3,5}$, 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 ($t_{1,5}$) oraz czas między poszczególnymi wiadomościami ($t_{3,5}$). Mimo bardzo zbliżonego znaczenia (dlatego prawdopodobnie FreeMODBUS nie wykorzystuje $t_{1,5}$), 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 $t_{1,5}$ oraz $t_{3,5}$. Drugie rozwiązanie ma dwie kluczowe zalety – oszczędność timerów oraz elastyczność. Ponieważ MODBUS może pracować przy różnych prędkościach przesyłu danych, także i czasy $t_{1,5}$ i $t_{3,5}$ są zmienne. Zamiast komplikować procedurę inicjalizacji 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 $50\mu\text{s}$ – 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:

```
1 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:

```
1 MB_RESPONSE_STATE MB_GetResponse(uint8_t addr, MB_FUNCTION f,  
2 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 oczekiwać 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 sugestię – 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:

4.2. TREŚĆ ĆWICZENIA

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

gdzie `write_single_coil_3` zdefiniowane jest jako:

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

W odpowiedzi otrzymane zostanie echo wiadomości nadanej:

```
1 uint8_t *resp;  
2 uint16_t resplen;  
3 MB_RESPONSE_STATE respstate;  
4 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 zmierzone (np. dyskretne wejście 4), należałoby wysłać następującą wiadomość:

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

gdzie

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

Odpowiedź urządzenia *slave* odbierana jest przy użyciu:

```
1 uint8_t *resp;  
2 uint16_t resplen;  
3 MB_RESPONSE_STATE respstate;  
4 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.

Proponowana implementacja 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 50µs,
- `void __attribute__((weak)) Enable50usTimer(void)` – funkcja służąca do włączenia działania timera odliczającego kwanty 50µs,
- `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) – jednocześnie 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 10µs,
- za odliczanie kwantów 50µs 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 zgłaszanemu 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 przerw TXE, RXNE, których funkcja obsługi zdefiniowana jest jako:

```
1 void USART1_IRQHandler(void){
2     if( USART_GetITStatus(USART1, USART_IT_RXNE) ){
3         USART_ClearITPendingBit(USART1, USART_IT_RXNE);
4         SetCharacterReceived(true);
5     }
6     if( USART_GetITStatus(USART1, USART_IT_TXE) ){
7         USART_ClearITPendingBit(USART1, USART_IT_TXE);
8         SetCharacterReadyToTransmit();
9     }
10 }
```

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:

```
1 void Communication_Put(uint8_t ch){
2     USART_SendData(USART1, ch);
3 }
4
5 uint8_t Communication_Get(void){
6     uint8_t tmp = USART_ReceiveData(USART1);
```



```

7  SetCharacterReceived(false);
8  return tmp;
9  }

```

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

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

```

1 void Communication_Mode(bool rx, bool tx){
2     USART_ITConfig(USART1, USART_IT_RXNE, rx?ENABLE:DISABLE);
3     USART_ITConfig(USART1, USART_IT_TXE, tx?ENABLE:DISABLE);
4 }

```

Kolejnymi funkcjami wymagającymi implementacji są:

```

1 void Enable50usTimer(void){
2     TIM_ITConfig(TIM4, TIM_IT_Update, ENABLE);
3 }
4
5 void Disable50usTimer(void){
6     TIM_ITConfig(TIM4, TIM_IT_Update, DISABLE);
7 }

```

które są wykorzystane do zatrzymywania i uruchamiania timera odmierzającego kwanty 50µs. Aby naliczanie 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.

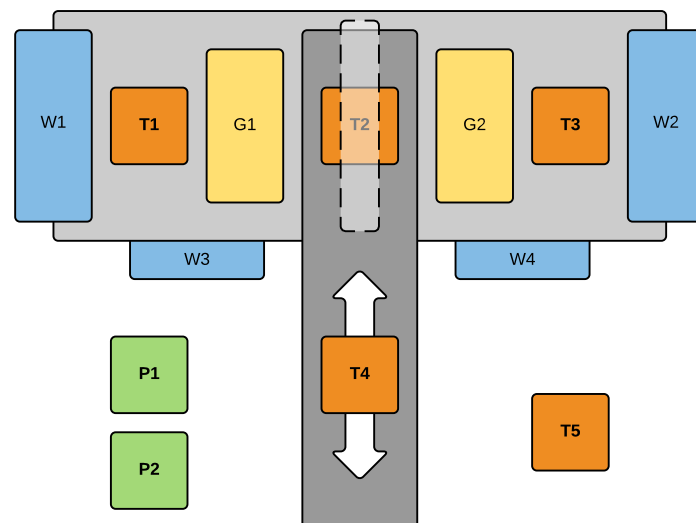
Procedura inicjalizacji komunikacji z użyciem protokołu MODBUS składa się z następujących etapów:

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.

4.2.6. Stanowisko grzejąco-chłodzące

W tym ćwiczeniu (oraz następnych) wykorzystane zostanie stanowisko grzejąco-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.



Rys. 4.4. Schemat rozmieszczenia elementów wykonawczych i pomiarowych w stanowisku grzejąco-chłodzącym

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. 4.4. 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 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 stopnia Celsjusza. Wartość 2500 znajdująca się w rejestrze typu *Input Register* oznacza, że związany 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 błędów należy przeanalizować w oparciu o dokumentację protokołu MODBUS RTU.

4.3. Wykonanie ćwiczenia

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 μs kwanty, inkrementując w ten sposób licznik.
- Korzystając z dostarczonych funkcji należy zrealizować prosty regulator, który będzie regulował temperaturę T1 stanowiska grzejąco-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 4-20mA i wynik pomiaru przedstawić na wyświetlaczu w °C (jako symbol ° 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.

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

5.1. Wprowadzenie

Celem tego ćwiczenia jest zapoznanie się z nową platformą z rodziny 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 przerw. 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).

5.2. Treść ćwiczenia

5.2.1. STM32F746G-DISCO

Przejsie z płytki rozwojowej ZL27ARM na płytkę STM32F746G-DISCO (Rys. 5.1) nie jest skomplikowanym procesem. Jednak należy pamiętać, że rodzina mikrokontrolerów STM32F7 i wyższe nie posiadają już dostosowanych do nich standardowych bibliotek peryferiów. Wyparta ona została przez bibliotekę HAL (*Hardware Abstraction Layer*), która w połączeniu z generatorem kodu CubeMX ma za zadanie usprawnić projektowanie



Rys. 5.1. Zdjęcie płytki rozwojowej STM32F746G-DISCO

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 płyta wyposażona jest w wyświetlacz LCD-TFT o przekątnej 4,3 cala, rozdzielczości 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 płycie 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 przydanym aspektem tej płytki 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.

5.2.2. Biblioteka HAL

W dotychczasowych projektach używana była Standardowa Biblioteka Peryferiów. Dla mikrokontrolerów z rodziny 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 peryferiów 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 ujednolicień w kodzie, twórcy i użytkownicy takiego kodu narażają się na pojawiające się redundancje. Tak jak biblioteka SPL nie jednokrotnie powtarza te same operacje na rejestrach, tak i biblioteka HAL generuje powtórzenia wywołań funkcji SPL. Tutaj należy zaznaczyć, że wiele funkcji ze Standardowej Biblioteki Peryferiów wchodzi w skład biblioteki HAL. Co więcej nierzadko biblioteka HAL wprowadza jedynie nową nazwę funkcji znanej 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.

5.2.3. 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:

```
1 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 jednoczesnych dotknięć ekranu,
- `touchX` – tablica przechowująca współrzędną poziomą dla każdego z wykrytych jednoczesnych dotknięć ekranu (początek osi znajduje się z lewej strony ekranu),
- `touchY` – tablica przechowująca współrzędną pionową dla każdego z wykrytych jednoczesnych 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 zgłębienie tego tematu pozostawia się zainteresowanym. W związku z uproszczoną obsługą ekranu dotykowego, zamiast

oczekiwania na przerwanie sygnalizujące zmianę stanu ekranu, będzie on regularnie odpytany 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.

5.2.4. 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.

5.2.5. Szablon projektu

Motywacją do napisania niniejszego dokumentu jest niemały poziom skomplikowania projektu na ćwiczenie wykorzystujące mikrokontroler STM32F746G. W tym dokumencie znajdują się informacje na temat dostępnych timerów, wykorzystanych przerwań oraz struktury projektu. Zostanie także krótko opisana kolejność wykonywania operacji, oraz możliwe problemy, które mogą wystąpić zarówno teraz jak i w przyszłych projektach.

Płytki rozwojowa

Wykorzystany mikrokontroler STM32F746G jest częścią płytki rozwojowej, na której zamontowany został z jednej strony wyświetlacz z ekranem dotykowym, a z drugiej takie elementy jak złącze do Arduino Uno, port Ethernet, złącze do kamery, złącze karty μ SD i wiele innych. Z naszego punktu widzenia najbardziej interesujące są: wyświetlacz, przycisk B1 (oznaczany również jako „User”) oraz dioda LD1. W przyszłości wykorzystane zostanie dodatkowo złącze Arduino Uno, aby zamontować tam dodatkowe przetworniki cyfrowo analogowe.

Wyświetlacz

Wyświetlacz podłączony do płytki rozwojowej jest wyświetlaczem kolorowym, który obsługiwany będzie przy użyciu biblioteki BSP (*Board Support Package*), pozwalającej na wygodną realizację podstawowych operacji. Do realizacji ćwiczenia nie jest wymagana znajomość modułu odpowiedzialnego za obsługę wyświetlacza, ani umiejętności jego konfiguracji, gdyż liczba parametrów jakie są dostępne w tym celu przekracza znacznie zakres tych ćwiczeń. Dodatkowo szablon dostępny jako punkt startowy ćwiczenia również nie został napisany „z palca”, a wygenerowany z użyciem STM32CubeMX – jest to znacznie szybsza metoda, choć nie koniecznie najlepsza.

Podstawowymi funkcjami, które będą wykorzystane dalszych ćwiczeniach są:

- rysowanie pikseli,
- rysowanie kształtów,
- rysowanie tekstu,
- zmiana koloru tła,
- zmiana koloru pierwszoplanowego,

5.2. TREŚĆ ĆWICZENIA

- pobranie koloru tła,
- pobranie koloru pierwszoplanowego.

Nazwy funkcji realizujących powyższe zadania są intuicyjne. Wszystkie funkcje związane z wyświetlaczem rozpoczynają się od `BSP_LCD_`. W tym momencie warto zauważyć, że dopełnianie składni (skrót **CTRL+SPACJA**) znacznie upraszcza przeglądanie dostępnych funkcji bibliotecznych. Funkcja rysująca piksele ma następującą deklarację (fragment z pliku `stm32746g_discovery_lcd.c`):

```
1 /**
2  * @brief Draws a pixel on LCD.
3  * @param Xpos: X position
4  * @param Ypos: Y position
5  * @param RGB_Code: Pixel color in ARGB mode (8-8-8-8)
6  * @retval None
7  */
8 void BSP_LCD_DrawPixel(uint16_t Xpos, uint16_t Ypos, uint32_t RGB_Code)
```

Jako pierwszy argument należy wstawić numer wiersza, jako drugi numer kolumny piksela, który ma zmienić kolor na podany w trzecim argumentcie. Numeracja wierszy i kolumn rozpoczyna się od 0, gdzie punkt (0,0) znajduje się w lewym górnym rogu wyświetlacza. Wszelkie punkty między wierszami 0 i 271 włącznie oraz między kolumnami 0 i 459 włącznie są widoczne na wyświetlaczu, Istnieje jednak zestaw punktów, które nie mają odzwierciedlenia w rzeczywistości i są wykorzystywane do innych celów – będą one ignorowane w trakcie ćwiczeń. Ostatnim argumentem funkcji jest kolor – wiele zostało już zdefiniowanych w pliku `stm32746g_discovery_lcd.h`:

```
1 #define LCD_COLOR_BLUE          ((uint32_t)0xFF0000FF)
2 #define LCD_COLOR_GREEN        ((uint32_t)0xFF00FF00)
3 #define LCD_COLOR_RED           ((uint32_t)0xFFFF0000)
4 #define LCD_COLOR_CYAN         ((uint32_t)0xFF00FFFF)
5 #define LCD_COLOR_MAGENTA       ((uint32_t)0xFFFF00FF)
6 #define LCD_COLOR_YELLOW        ((uint32_t)0xFFFFFF00)
7 #define LCD_COLOR_LIGHTBLUE     ((uint32_t)0xFF8080FF)
8 #define LCD_COLOR_LIGHTGREEN    ((uint32_t)0xFF80FF80)
9 #define LCD_COLOR_LIGHTRED      ((uint32_t)0xFFFF8080)
10 #define LCD_COLOR_LIGHTCYAN     ((uint32_t)0xFF80FFFF)
11 #define LCD_COLOR_LIGHTMAGENTA  ((uint32_t)0xFFFF80FF)
12 #define LCD_COLOR_LIGHTYELLOW   ((uint32_t)0xFFFFFF80)
13 #define LCD_COLOR_DARKBLUE      ((uint32_t)0xFF000080)
14 #define LCD_COLOR_DARKGREEN     ((uint32_t)0xFF008000)
15 #define LCD_COLOR_DARKRED       ((uint32_t)0xFF800000)
16 #define LCD_COLOR_DARKCYAN     ((uint32_t)0xFF008080)
17 #define LCD_COLOR_DARKMAGENTA   ((uint32_t)0xFF800080)
18 #define LCD_COLOR_DARKYELLOW    ((uint32_t)0xFF808000)
19 #define LCD_COLOR_WHITE         ((uint32_t)0xFFFFFFFF)
20 #define LCD_COLOR_LIGHTGRAY     ((uint32_t)0xFFD3D3D3)
21 #define LCD_COLOR_GRAY          ((uint32_t)0xFF808080)
22 #define LCD_COLOR_DARKGRAY      ((uint32_t)0xFF404040)
23 #define LCD_COLOR_BLACK         ((uint32_t)0xFF000000)
24 #define LCD_COLOR_BROWN        ((uint32_t)0xFFA52A2A)
25 #define LCD_COLOR_ORANGE        ((uint32_t)0xFFFFA500)
26 #define LCD_COLOR_TRANSPARENT   ((uint32_t)0xFF000000)
```

Jak widać kolory zostały zapisane jako zmienne `uint32_t` – tak należy również je przechowywać w swoim kodzie. Zapis ten wynika z faktu, iż kolory w ramach tego i następnych ćwiczeń reprezentowane będą dla ułatwienia jako liczby, gdzie najbardziej znaczące 8 bitów określa przeźroczystość (0x00 – całkowicie przeźroczysty, 0xFF – całkowicie widoczny), następne 8 bitów określa natężenie koloru czerwonego (0x00 – zerowe, 0xFF – maksymalne), kolejne 8 bitów określa natężenie koloru zielonego (wartości analogicznie jak w przypadku koloru czerwonego), a najmłodszych 8 bitów określa natężenie koloru niebieskiego (analogicznie jak dla czerwonego i zielonego). Przeznaczenie poszczególnych bajtów (8

bitów) jest wyraźnie widoczne w przypadku kolorów `LCD_COLOR_RED`, `LCD_COLOR_GREEN` oraz `LCD_COLOR_BLUE`. Mimo zastosowania zapisu wykorzystującego przeźroczystość, kolor „przeźroczysty” jest równoważny kolorowi czarnemu. Wynika to z faktu iż korzystamy wyłącznie z jednej warstwy – zakładamy więc, że pod warstwą roboczą znajduje się czarna otchłań. A więc narysowanie białego prostokąta o przeźroczystości na poziomie 50% spowoduje narysowaniem szarego prostokąta. W szczególności nie spowoduje to połowicznego przykrycia obecnie wytworzonego obrazu białym prostokątem! Wniosek z powyższego jest raczej oczywisty – nie ma potrzeby i sensu operować przeźroczystością w ramach tego i następnych ćwiczeń.

Skoro wiadomo już jak rysować pojedyncze piksele, warto rozważyć bardziej ambitne kształty. Konwencja nazewnicza jest następująca: początek `BSP_LCD_Draw` oznacza wyrysowanie wyłącznie krawędzi kształtu, natomiast `BSP_LCD_Fill` spowoduje wypełnienie również wnętrza. W obu przypadkach użyty zostanie kolor pierwszoplanowy. Dostępne kształty to: `Circle` (koło/okrąg), `Ellipse` (elipsa), `Polygon` (wielokąt) oraz `Rect` (prostokąt). Argumenty dotyczące tych funkcji są oczywiste – w razie jednak problemów warto zajrzeć do komentarza nad definicją danej funkcji. Poza powyższymi kształtami, które mogą mieć wypełnienie, jest możliwość wyrysowania kształtów takie jak: `Line` (linia), `VLine` (linia pionowa), `HLine` (linia pozioma), `Pixel` (piksel/punkt), `Bitmap` (bitmapa). Jak wcześniej – argumenty powinny być zrozumiałe, w razie jednak problemów dokładny opis znajduje się tuż nad definicją danej funkcji. Warto zauważyć, że niektóre funkcje nie rysują dokładnie tego, czego można by się było spodziewać. Np. funkcja `BSP_LCD_DrawRect` rysuje prostokąt, lecz w jego prawym dolnym rogu brakuje narożnego piksela.

Zdefiniowany został dodatkowo zestaw funkcji do wyświetlania znaków. Funkcje te mają wspólny początek `BSP_LCD_Display`, natomiast końcówki mogą być następujące: `Char` (znak), `StringAt` (ciąg znaków w wybranym miejscu), `StringAtLine` (ciąg znaków w wybranej linii). Wyświetlanie znaku wymaga podania pozycji w pionie i poziomie oraz znaku do wyświetlenia. Warto zauważyć, że podana pozycja jest lewym górnym rogiem rysowanego znaku (wraz z jego tłem). Funkcja `StringAt` jako ostatni argumenty przyjmuje ”tryb pracy”. Zdefiniowane są one następująco:

- `LEFT_MODE` – lewy górny róg napisu umieszczany jest w podanym punkcie początkowym, punkt (0,0) znajduje się w lewym górnym rogu ekranu,
- `RIGHT_MODE` – prawy górny róg napisu umieszczany jest w podanym punkcie początkowym, punkt (0,0) znajduje się w prawym górnym rogu ekranu,
- `CENTER_MODE` – środek górnej krawędzi napisu umieszczany jest w podanym punkcie początkowym, punkt (0,0) znajduje się na środku górnej krawędzi ekranu.

Funkcja `StringAtLine` zawsze wypisuje tekst począwszy od lewej krawędzi ekranu (między krawędzią a tekstem jest 1 piksel odstępu). W zależności od wybranej czcionki ekran dzielony jest na linie, w których wypisywany jest tekst. Numer linii podawany jest jako pierwszy argument – linie numerowane są od zera.

Do przydatnych funkcji należą także te, które nie generują obrazu – modyfikują one jednak produkty następujących po nich operacji. Są to:

- `BSP_LCD_SetTextColor` – ustawia kolor pierwszoplanowego (zarówno dla tekstu jak i figur),
- `BSP_LCD_SetBackColor` – ustawia kolor tła dla tekstu,
- `BSP_LCD_GetTextColor` – zwraca kolor pierwszoplanowy,
- `BSP_LCD_GetBackColor` – zwraca kolor tła,
- `BSP_LCD_ClearStringLine` – czyści podaną linię tekstu kolorem tła,

- `BSP_LCD_Clear` – czyści cały ekran podanym kolorem,
- `BSP_LCD_GetFont` – zwraca wskaźnik na strukturę definiującą czcionkę (zawiera między innymi wysokość i szerokość znaku),
- `BSP_LCD_GetXSize` – zwraca wysokość ekranu,
- `BSP_LCD_GetYSize` – zwraca szerokość ekranu,
- `BSP_LCD_SetFont` – ustawia czcionkę (jako argument warto wykorzystać zdefiniowane w bibliotece struktury o nazwach `Font8`, `Font12`, `Font16`, `Font20` oraz `Font24`, gdzie liczba stojąca na końcu nazwy oznacza wysokość znaku w danej czcionce).

Pozostałe funkcje są raczej nieprzydatne z punktu widzenia ćwiczenia – najczęściej wykorzystywane są w ramach inicjalizacji i konfiguracji wyświetlacza.

Ekran dotykowy

Ekran dotykowy wykorzystany w tym i następnych ćwiczeniach zostanie jako główne narzędzie do interakcji z użytkownikiem. Do realizacji tego zadania wykorzystana zostanie funkcja `BSP_TS_GetState`. Jako argument przyjmuje ona wskaźnik na strukturę typu `TS_StateTypeDef`. Po wywołaniu funkcji `BSP_TS_GetState`, w podanej strukturze znajdują się informacje dotyczące obecnego stanu ekranu dotykowego. Najważniejsze i prawdopodobnie jedyne przydatne w ramach tych ćwiczeń pola tej struktury to:

- `touchDetected` – liczba wykrytych dotknięć,
- `touchX` – tablica współrzędnych `x` poszczególnych wykrytych dotknięć (indeksowana od 0 do `N`, gdzie `N` to liczba wykrytych dotknięć),
- `touchY` – tablica współrzędnych `y` poszczególnych wykrytych dotknięć (indeksowana od 0 do `N`, gdzie `N` to liczba wykrytych dotknięć).

Istnieje możliwość wyzwolenia przerwania w momencie wykrycia zmiany stanu ekranu dotykowego, lecz dla ułatwienia obsługi zostanie ona zastąpiona regularnym, częstym odpytywaniem (proponowane jest odpytywanie co 100ms).

Przycisk B1

Na płycie uruchomieniowej został zamontowany przycisk. Znajduje się on na spodniej stronie płytki – po przeciwnej stronie niż lewy górny róg ekranu (patrząc od spodu płytki jest to prawy górny róg). Odczyt stanu przycisku realizowany jest przy użyciu funkcji `BSP_PB_GetState(BUTTON_TAMPER)`. Rezultatem wywołania tej funkcji jest wartość określająca czy przycisk został wciśnięty (`GPIO_PIN_SET`) czy nie (`GPIO_PIN_RESET`). Dla ułatwienia korzystania z tego przycisku został zrealizowany na płycie rozwojowej filtr dolno-przepustowy, a więc jeden z klasycznych mechanizmów niwelacji drgań styków. Szczegóły znajdują się na rysunku 21 w dokumencie UM1907.

Dioda LD1

W przeciwieństwie do płyty ZL27ARM, tutaj zamontowano wyłącznie jedną diodę do użytku programisty. Jest to zielona dioda LD1 – znajduje się ona pod czarnym przyciskiem Reset, który znajduje się bezpośrednio pod przyciskiem B1. Sterować nią można za pomocą funkcji `BSP_LED_Off(LED1)` – wyłączenie diody, `BSP_LED_On(LED1)` – włączenie diody oraz `BSP_LED_Toggle(LED1)` – przełączenie diody. Może być ona zarówno wygodnym mechanizmem do prostego debugowania programu, lub kontrolką informującą o „zajętości” systemu.

Timery

Do dyspozycji użytkownik tego mikrokontrolera ma wiele timerów: zaawansowane TIM1 i TIM8, ogólnego użytku TIM2-TIM5 oraz TIM9-TIM14, a także podstawowe TIM6 i TIM7. Aby nie zagłębiać się w szczegóły konfiguracji timerów – część z nich została przykładowo skonfigurowana. Zrealizowane to zostało dla timerów TIM2-TIM5. Ich konfiguracja znajduje się w pliku `main.c` w funkcjach o nazwach `MX_TIMx_Init`, gdzie za `x` należy wstawić numer timera. Metoda konfiguracji różni się nieznacznie od tej stosowanej w ramach standardowej biblioteki peryferiali, lecz nazwy pól struktury inicjalizującej w większości się pokrywają. W ramach ćwiczeń będzie konieczne skonfigurowanie okresu wyzwolenia przerwania związanego z przepełnieniem timera. Tak jak w SPL wymagało to odpowiedniego uzupełnienia wartości `Period` oraz `Prescaler`, tak samo i tutaj należy odpowiednio zmodyfikować te pola. Ich znaczenie jest identyczne jak w przypadku SPL. Należy zwrócić uwagę jednak na pewien szczegół – dla timerów TIM2 i TIM5 pole `Period` jest 32-bitowe, natomiast dla timerów TIM3 i TIM4 jest ono 16-bitowe.

Funkcje do obsługi przerw również już są zdefiniowane (znajdują się w pliku `stm32f7xx_it.c`) i zgodnie z konwencją z SPL mają nazwy od `TIM2_IRQHandler` po `TIM5_IRQHandler`. Każda z nich zawiera wywołanie `HAL_TIM_IRQHandler(&htimx);`, gdzie za `x` należy podstawić numer timera. Służy ono do odpowiedniej obsługi przerwania – w szczególności wyczyszczenia odpowiednich bitów przerwania oczekującego i wywołania funkcji `Callback`, związanej z odpowiednim zdarzeniem. Dzięki takiemu schematowi programista może nie implementować już całego przerwania, a jedynie funkcję związaną z interesującym go zdarzeniem. Dla przykładu, obsługę przepełnienia rejestru timera TIM2 można zrealizować jako:

```
1 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
2     if(htim->Instance == TIM2){
3         // obsługa przerwania
4     }
5 }
```

gdzie funkcja obsługi przerwania timera TIM2 wygląda następująco:

```
1 void TIM2_IRQHandler(void){
2     HAL_TIM_IRQHandler(&htim2);
3 }
```

Alternatywnie można oczywiście zastosować podejście klasyczne, tj. sprawdzenie flagi i własnoręczne wyczyszczenie stosownego bitu oczekującego przerwania, lecz mocno zalecane jest zaimplementowanie funkcji `HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *tim)`, zgodnie z powyższym wzorem. Funkcję tę można zaimplementować również w pliku `main.c` co pozwoli na ograniczenie zamieszania związanego ze zmiennymi o modyfikatorze `extern`.

Priorytety przerw ustawione są w funkcji `main` w pliku `main.c`:

```
1  /* Timers initialisation */
2  MX_TIM2_Init();
3  MX_TIM3_Init();
4  MX_TIM4_Init();
5  MX_TIM5_Init();
6
7  /* Interrupts configuration */
8  HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0); // priority of SysTick
9  HAL_NVIC_SetPriority(TIM2_IRQn, 1, 0); // priority of TIM2
10 HAL_NVIC_SetPriority(TIM3_IRQn, 0, 0); // priority of TIM3
11 HAL_NVIC_SetPriority(TIM4_IRQn, 0, 0); // priority of TIM4
12 HAL_NVIC_SetPriority(TIM5_IRQn, 0, 0); // priority of TIM5
13
14 /* Interrupts enabling */
15 HAL_NVIC_EnableIRQ(TIM2_IRQn); // enabling handling functions of TIM2
```

```

16 HAL_NVIC_EnableIRQ(TIM3_IRQn); // enabling handling functions of TIM3
17 HAL_NVIC_EnableIRQ(TIM4_IRQn); // enabling handling functions of TIM4
18 HAL_NVIC_EnableIRQ(TIM5_IRQn); // enabling handling functions of TIM5

```

Kolejno zrealizowana jest inicjalizacja (konfiguracja) samych timerów, następnie ustawienie ich priorytetów (oraz SysTick-a) a na koniec włączenie samych funkcji do obsługi przerwań. O ile ta ostatnia różni się od jej wersji z SPL jedynie dodanym początkiem HAL_, o tyle funkcja do ustawienia priorytetów nie wykorzystuje już specjalnej struktury do konfiguracji. Argumentami funkcji HAL_NVIC_SetPriority są kolejno: nazwa przerwania, priorytet wyłączenia oraz podpriorytet. Ważne jest, aby pamiętać, że priorytety domyślnie ustawione są tak, że **wszystkie 4 bity opisują priorytet wyłączenia**, nie pozostawiając nic na podpriorytet.

Przetwornik analogowo-cyfrowy

W ramach tego i następnych ćwiczeń wykorzystany został przetwornik analogowo cyfrowy powiązany z DMA (*Direct Memory Access*). Przekłada się to na kod programu w taki sposób, że student nie musi już obsługiwać przerwania związanego z końcem przetwarzania. Wynik przetwarzania jest nieustannie aktualizowany i zawiera się w zmiennej adc_value. Jest to wynik uśredniony ze 100 ostatnich pomiarów. Implementacja uśredniania została zrealizowana w funkcji (w pliku main.c)

```

1 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* AdcHandle){
2     static int i=0;
3     static uint32_t tmpval= 0;
4     for(i=0,tmpval=0;i<ADC_BUFFER_LENGTH; ++i)
5         tmpval += uhADCxConvertedValue[i];
6     adc_value = tmpval/ADC_BUFFER_LENGTH;
7 }

```

gdzie ADC_BUFFER_LENGTH jest zdefiniowany w pliku main.c i wynosi 100, natomiast tablica

uhADCxConvertedValue zawiera zrealizowane pomiary – właśnie tutaj informacje z ADC przekazuje DMA. Funkcja HAL_ADC_ConvCpltCallback wywoływana jest w momencie, gdy DMA wypełni wszystkie 100 elementów wspomnianej tablicy. W tym momencie dokonywane jest powyższe uśrednianie i wynik zapisywany jest do adc_value.

5.2.6. Fraktal

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

```

1 void mandelbrot(){
2     static int Px = 0;
3     static int Py = 0;
4     static float x = 0;
5     static float xtemp = 0;
6     static float y = 0;
7     static float x0 = 0;
8     static float y0 = 0;
9     static uint8_t iteration = 0;
10
11     // obszar rysowany zbioru Mandelbrota
12     static float xmin = -0.1011-0.01;
13     static float xmax = -0.1011+0.01;
14     static float ymin = 0.9563-0.01;
15     static float ymax = 0.9563+0.01;
16
17     // for(Px=nr pierwszej kolumny pixeli; Px<=nr ostatniej kolumny pixeli; ++Px)
18     for(;;++Px){ // TODO

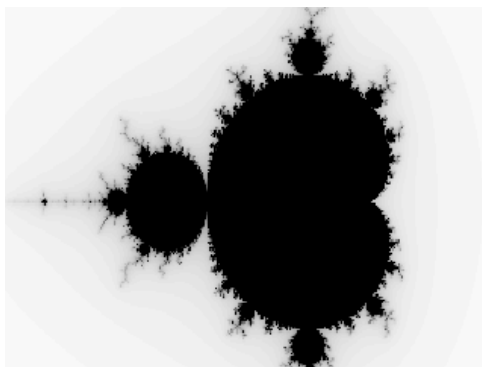
```

```

19 // for(Py=nr pierwszej wiersza pikseli; Px<=nr ostatniego wiersza pikseli; ++Py)
20 for(;;++Py){ //TODO
21     // wyznaczyć x0 i y0 tak, aby:
22     // x0 = xmin <=> Px = indeks pierwszej kolumny pikseli;
23     // x0 = xmax <=> Px = indeks ostatniej kolumny pikseli;
24     // y0 = ymin <=> Py = indeks pierwszej wiersza pikseli;
25     // y0 = ymax <=> Py = indeks ostatniego wiersza pikseli;
26
27     x0 = 0.0; // TODO
28     y0 = 0.0; // TODO
29     // algorytm rysujący zbiór Mandelbrota
30     x = 0.0;
31     y = 0.0;
32     iteration = 0;
33     while ((x*x + y*y < 2*2) && (iteration < 0xFF)) {
34         xtemp = x*x - y*y + x0;
35         y = 2*x*y + y0;
36         x = xtemp;
37         iteration = iteration + 1;
38     }
39     // rysowanie piksela w punkcie Px, Py,
40     // o kolorze zależnym od wartości iteration (0x00-0xFF)
41     // TODO
42 }
43 }
44 }

```

Ponieważ umiejscowienie rysunku nie jest jednoznacznie zdefiniowane – do programisty należy zadanie odpowiedniego przeskalowania obszaru rysowania (wyznaczenie wartości x_0 i y_0) 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. 5.2.



Rys. 5.2. 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.

5.3. Wykonanie ćwiczenia

W ramach tego ćwiczenia 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 być przyciskiem modyfikującym obraz wyświetlany w odpowiadającej mu części:
 - lewa połowa ekranu powinna być przyciskiem bistabilnym (tj. jego stan przełącza się z każdym jego naciśnięciem) – jego przełączenie powinno powodować narysowanie fraktalu w dwóch różnych schematach kolorystycznych, tj. np. kolorach od czarnego do zielonego i od białego do czerwonego,
 - prawa połowa ekranu powinna być przyciskiem monostabilnym i jego wciśnięcie powinno powodować wyczyszczenie wykresu i rozpoczęcie ponownego wyświetlania nowych danych tak, jak po starcie programu (ekran ten powinien być czyszczony tak długo jak wciśnięty jest ten przycisk),
- 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 pisanie tej (nudnej) części programu.

6. Projekt 1: Implementacja algorytmów regulacji PID i DMC prostego procesu dynamicznego, interfejs użytkownika, archiwizacja pomiarów, dobór nastaw algorytmów, badania porównawcze

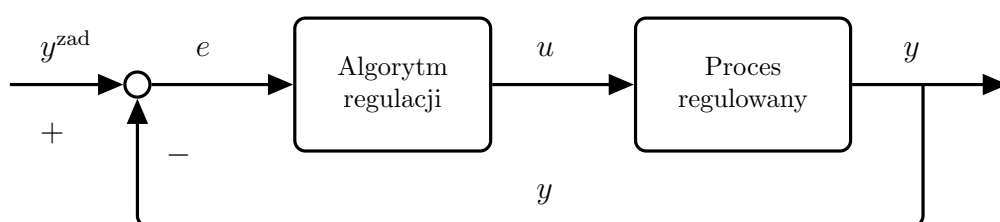
6.1. Wprowadzenie

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 przerwaniem 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.

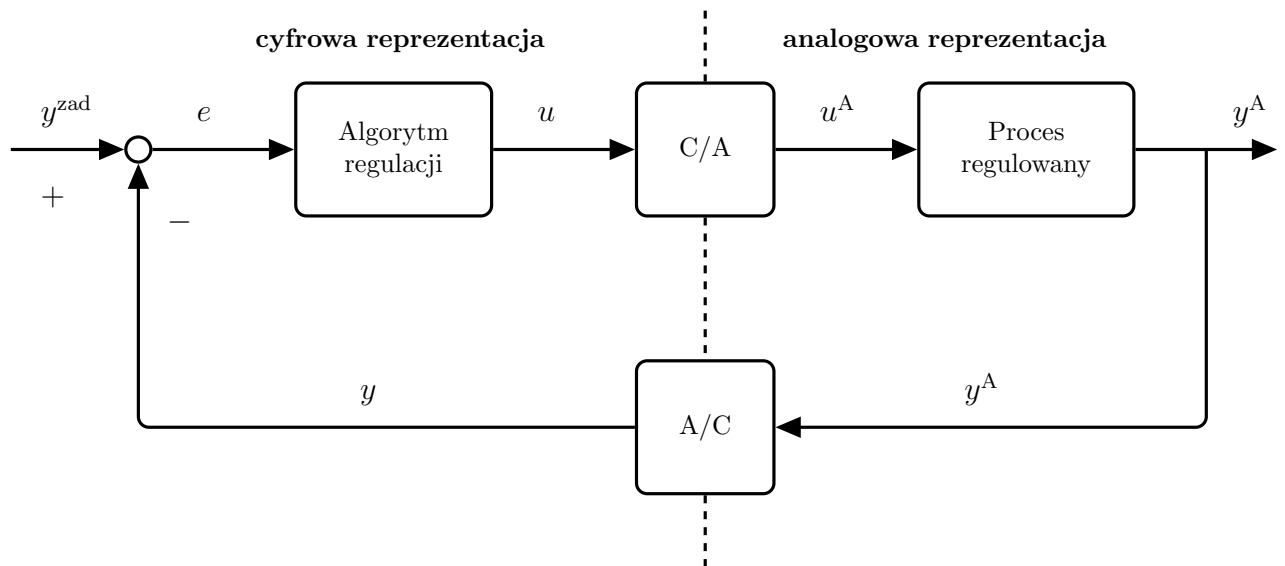
6.2. Treść projektu

6.2.1. 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 $y^{\text{zad}} - y$ oznacza się symbolem e (Rys. 6.1). 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 nieliniowe, aby na ich podstawie wyznaczać nowe sygnały sterowania. Rosnący poziom skomplikowania



Rys. 6.1. Schemat ideowy układu regulacji



Rys. 6.2. Schemat ideowy układu regulacji z uwzględnieniem przetworników analogowo-cyfrowych i cyfrowo-analogowych

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 satysfakcjonująca.

Realizując to ćwiczenie warto mieć świadomość istnienia przetworników analogowo-cyfrowych i cyfrowo-analogowych, które występują pomiędzy procesem regulowanym a regulatorem. Dodatkowo konieczne 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 funkcji do rozwiązywania zadań programowania kwadratowego).

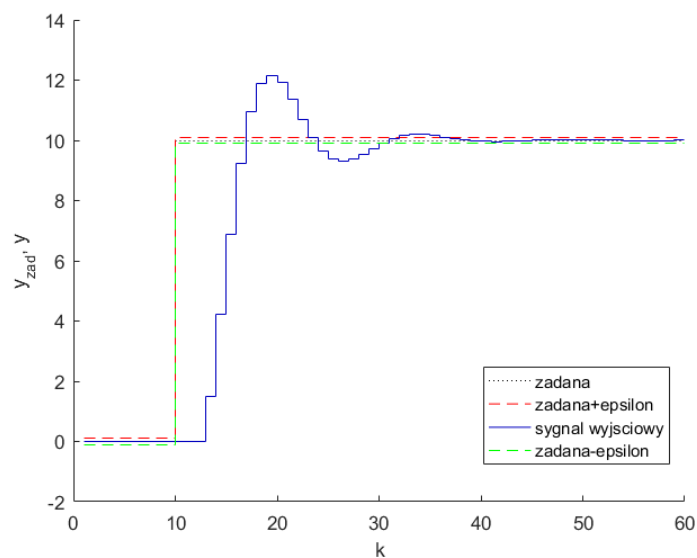
Zanim omówione zostaną algorytmy regulacji potrzebne jest wprowadzenie dwóch pojęć, które służą do oceny jakości regulacji. Są to „przesterowanie” oraz „czas ustalenia”. Obie te wartości są związane z odpowiedzią obiektu na skok wartości zadanej. Aby te wartości można było ze sobą porównać, a co za tym 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

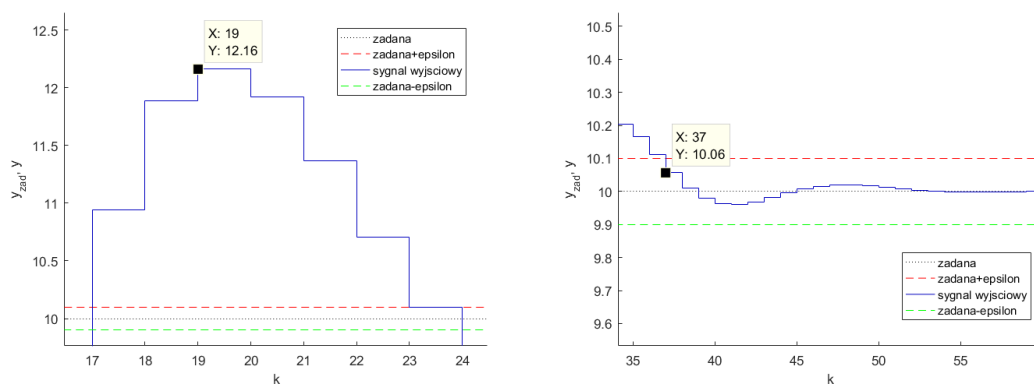
$$K_o = \frac{y_m - y^{zad}}{y^{zad}} \cdot 100\%$$

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 zadanego, do chwili gdy wartość bezwzględna uchybu przestaje przekraczać 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 $y^{zad} - \varepsilon$ do $y^{zad} + \varepsilon$ i więcej go nie opuszcza. Na Rys. 6.3 przedstawiony został wykres wartości wyjściowej obiektu regulacji, na podstawie którego wyznaczone mogą zostać przesterowanie oraz czas ustalenia. Rys. 6.4 (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. 6.4 (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.



Rys. 6.3. Wykres będący efektem eksperymentu, mający na celu wyznaczenie wartości przesterowania oraz czasu ustalenia



Rys. 6.4. Odczyt maksymalnej wartości sygnału wyjściowego obiektu regulacji będący podstawą do wyznaczenia przesterowania (lewy wykres) oraz odczyt czasu ustalenia (prawy wykres)

6.2.2. 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 związanym 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 dyskretny – 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 dyskretny k) algorytmu PID przedstawia się następująco:

$$u(k) = u_P(k) + u_I(k) + u_D(k) \quad (6.1)$$

gdzie

$$\begin{aligned} u_P(k) &= K e(k) \\ u_I(k) &= u_I(k-1) + \frac{K}{T_I} T \frac{e(k-1) + e(k)}{2} \\ u_D(k) &= K T_D \frac{e(k) - e(k-1)}{T} \end{aligned} \quad (6.2)$$

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 (różnicy między wyjściem procesu regulowanego a wartością zadaną) w dyskretny 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 – wzmocnienie. 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. $T_D = 0$), 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

uchyby, 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 wzmocnienia poszczególnych członów zgodnie z poniższym:

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

$$K_D = KT_D$$

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

We wzorze (6.2) 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

$$u(k) = r_2 e(k-2) + r_1 e(k-1) + r_0 e(k) + u(k-1) \quad (6.3)$$

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 (6.1) oraz (6.3) są sobie równoważne, tj. można z jednego przejść do drugiego wyłącznie przy użyciu prostych przekształceń (wystarczy od każdej strony równania (6.1) 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 „zmiennie, które stoją przy kolejnych uchybach”. Do realizacji ćwiczenia warto posłużyć się wzorem (6.1), 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:

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

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

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

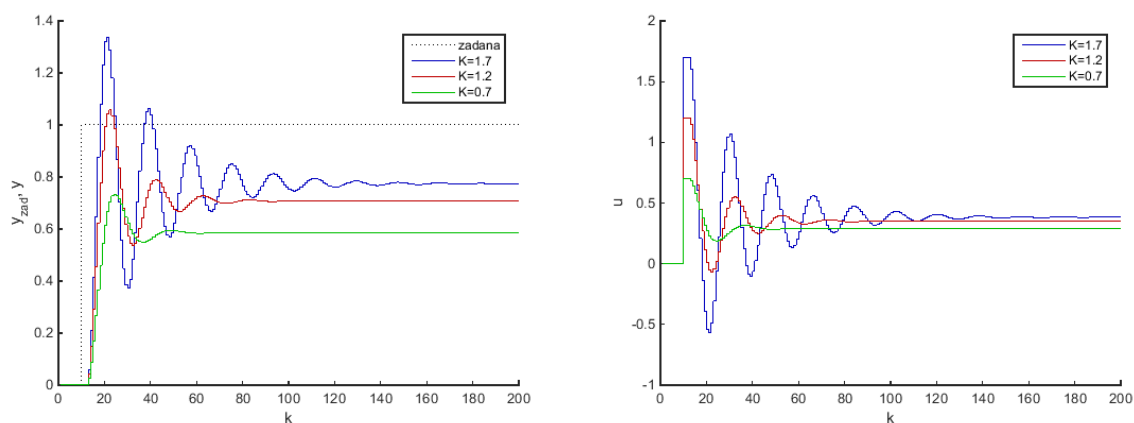
uwzględniając ewentualne wyłączanie poszczególnych członów.

Regulator P (wyłączone człony całkujący i różniczkujący) generuje sygnał sterujący proporcjonalny do różnicy między wartością zadaną a wyjściem procesu regulowanego. Na Rys. 6.5 widać, że, nawet po bardzo długim czasie, wyjście procesu regulowanego nie jest równe wartości zadanej. Wręcz przeciwnie – różnica między nimi jest stała. Zjawisko

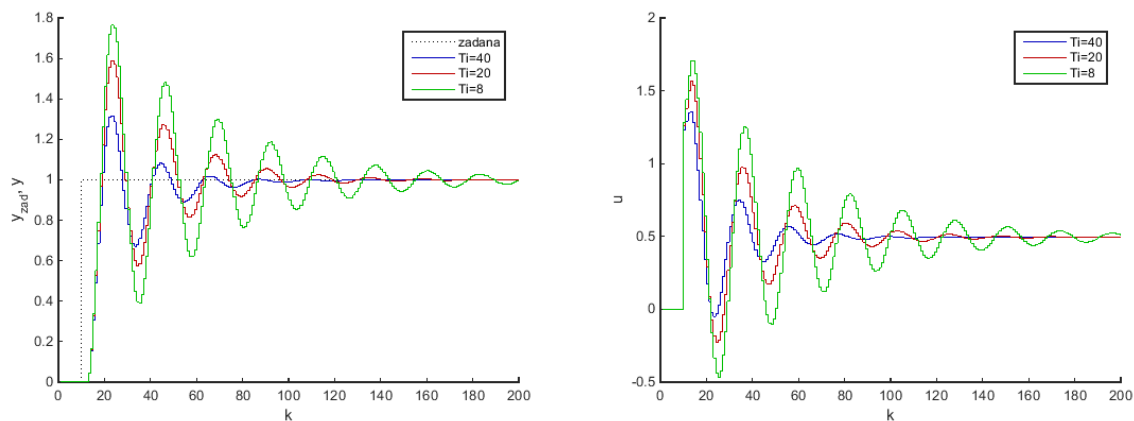
takie nazywane jest uchybem ustalonym. Aby zrozumieć to zjawisko warto zastanowić się co dziełoby się w sytuacji gdyby jednak uchyb był zerowy. Wtedy sterowanie również byłoby zerowe, a co za tym idzie wartość wyjściowa procesu by spadła. To spowodowałoby narastanie uchybu, który powodowałoby narastanie 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 wzmocnienia (nie ma tak dużego wzmocnienia, 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. 6.6. 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. 6.7 ma miejsce w chwili 20. W tym miejscu na wykresie sygnału sterującego zauważyć można bardzo wysoki skok wartości sterowania – jest on spowodowany właśnie przez człon różniczkujący, ponieważ różnica



Rys. 6.5. Wyjście oraz wejście procesu regulowanego – regulator P



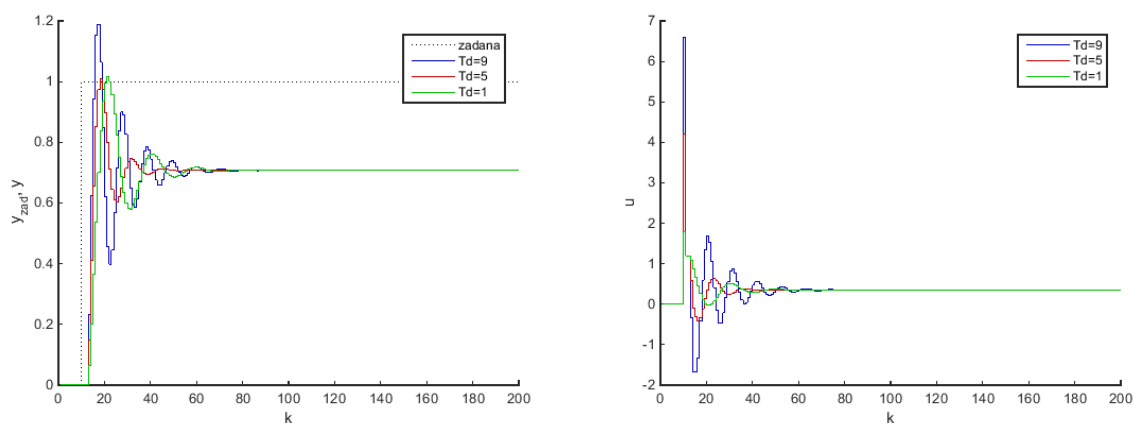
Rys. 6.6. Wyjście oraz wejście procesu regulowanego – regulator PI

między uchybem obecnym, a uchybem z poprzedniej chwili jest bardzo wysoka. Ponieważ rozważanym regulatorem jest regulator PD, ponownie można zaobserwować uchyb ustalony.

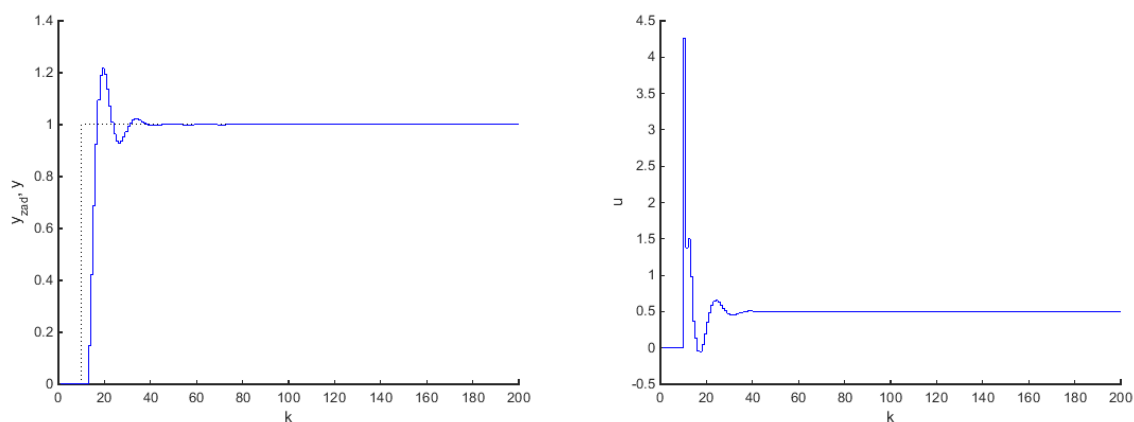
Regulator PID jest złączeniem wszystkich powyższych członów, a więc posiada wszystkie ich cechy. Najważniejszymi są brak uchybu ustalonego oraz chwilowy skok wartości sterowania w momencie zmiany wartości zadanej. Wykres pełnego regulatora PID widoczny jest na Rys. 6.8.

Na Rys. 6.9, 6.10 oraz 6.11 pokazane są przebiegi sygnałów wejściowego i wyjściowego w zależności od zmiany jednego z parametrów (nastaw) regulatora PID. Dla Rys. 6.10 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 pozostałego czasu.

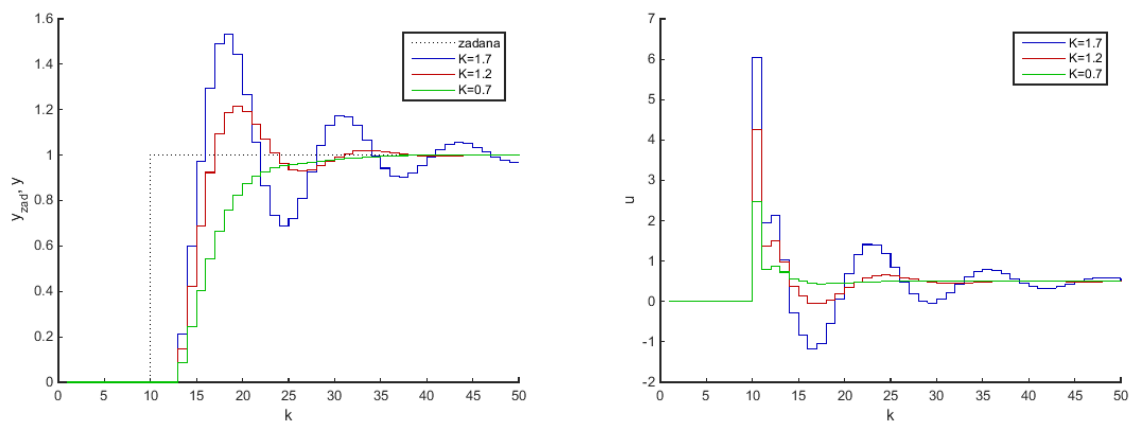
Ponieważ w prawie sterowania algorytmu PID nie ma miejsca na uwzględnienie ograniczeń na wartości sterowania, należy poradzić sobie z tym problemem osobno. Brak uwzględnienia ograniczeń powoduje, że gdy zostanie ono osiągnięte, a błąd będzie niezerowy, realizowane będzie niepotrzebne całkowanie, tj. wyznaczona wartość sygnału ste-



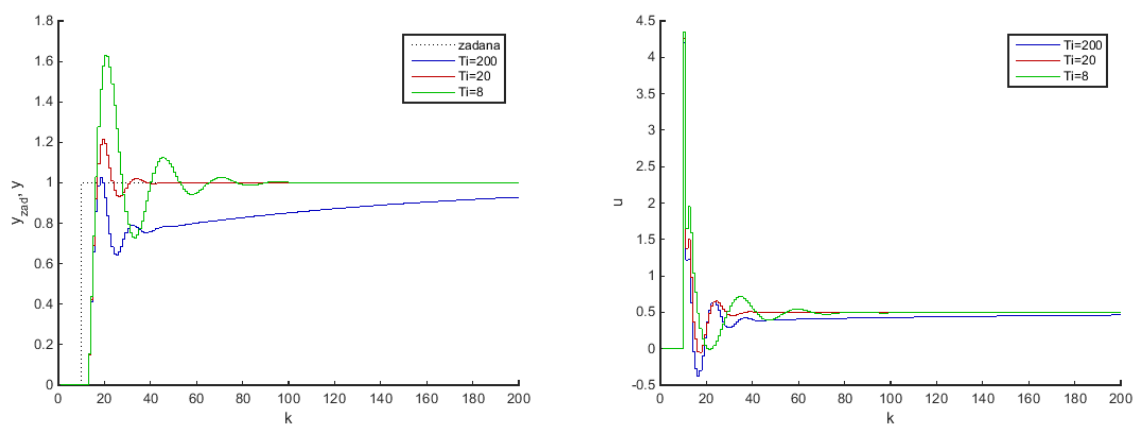
Rys. 6.7. Wyjście oraz wejście procesu regulowanego – regulator PD



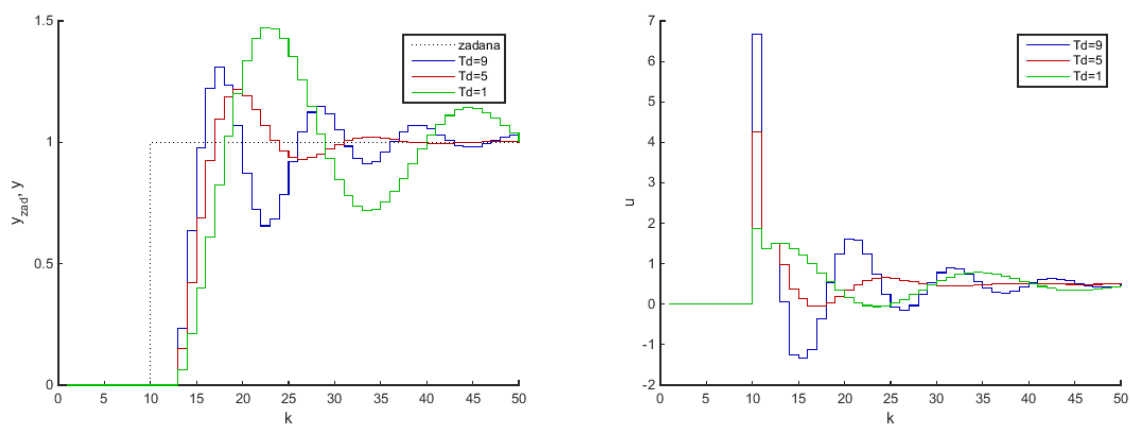
Rys. 6.8. Wyjście oraz wejście procesu regulowanego – regulator PID będący punktem odniesienia



Rys. 6.9. Wyjście oraz wejście procesu regulowanego w zależności od wzmocnienia – regulator PID



Rys. 6.10. Wyjście oraz wejście procesu regulowanego w zależności od czasu zdwojenia – regulator PID



Rys. 6.11. Wyjście oraz wejście procesu regulowanego w zależności od czasu wyprzedzenia – regulator PID

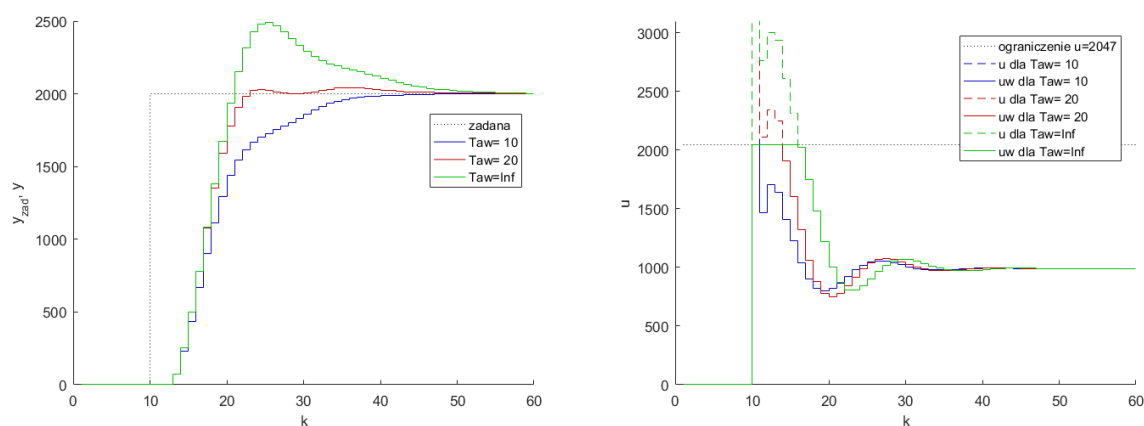
rującego będzie rosła mimo, że faktyczna maksymalna wartość sygnału sterującego już została osiągnięta. Efekty tego zjawiska widoczne są szczególnie w momencie, gdy sygnał wyjściowy przekroczy wartość zadaną – wtedy przez długi czas dokonywane jest „odcałkowanie”, tj. sygnał sterujący jest nieustannie 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 proporcjonalnego (6.2) do różnicy między faktycznym sygnałem sterującym a oczekiwanym

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

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. 6.12 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 jednocze-



Rys. 6.12. Wyjście oraz wejście procesu regulowanego – algorytm PID z oraz bez algorytmu *anti-windup* (pokazane wartości sygnału sterującego ograniczone są do 3100 dla zwiększenia czytelności)

śnie szybszą reakcję algorytmu regulacji w przypadku osiągnięcia ograniczeń i najczęściej również ograniczenie przesterowania.

6.2.3. 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 wzmocnienia regulatora K , aby sygnał wyjściowy obiektu regulacji miał charakter oscylacyjny. Trzeba tak dobrać wzmocnienie, 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 wzmocnieniem 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 rozsądną 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.

6.2.4. 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 wzmocnieniem 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 satysfakcjonujący 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 satysfakcjonujących wyników kończy procedurę strojenia.

6.2.5. 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 dyskretniej chwili (iteracji algorytmu) wyznaczany jest wektor przyszłych przyrostów wartości sterowania

$$\Delta U(k) = \begin{bmatrix} \Delta u(k|k) \\ \vdots \\ \Delta u(k + N_u - 1|k) \end{bmatrix} \quad (6.4)$$

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 zadanych wartości sygnału wyjściowego) a predykowaną trajektorią sygnału wyjściowego $\hat{y}(k + p|k)$ (tj. wektorem jego 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

$$\min_{\Delta U(k)} \left\{ \|Y^{\text{zad}}(k) - \hat{Y}(k)\|_{\mathbf{I}}^2 + \|\Delta U(k)\|_{\Lambda}^2 \right\} \quad (6.5)$$

gdzie $\Lambda > 0$, a $\|x\|_Y^2 = x^T Y x$. Macierz \mathbf{I} jest macierzą identycznościową (tj. diagonalną z samymi jedynkami na diagonalu 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 diagonalu 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 wyznaczyć wartość obecnego sterowania dodać do ostatniej wartości sygnału sterowania, tj. $u(k) = \Delta u(k|k) + u(k-1)$. W następnej iteracji algorytmu regulacji, uaktualniane 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 tę samą lub zbliżoną co ostatni element odpowiedzi skokowej. Wprowadzić więc warto 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 liniowość 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 (6.5) można wyznaczyć wprost jako

$$\Delta U(k) = \mathbf{K} [Y^{\text{zad}}(k) - Y^0(k)] \quad (6.6)$$

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

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

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 $u(k|k)$

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

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

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

6.2.6. Symulowany obiekt

Obiekt regulacji jest symulowany przy użyciu płytki rozwojowej STM32F746G-DISCOVERY. Jest ona tak samo wyposażona jak płytka działająca jako regulator, którą programować będzie student. Schemat połączenia jest przedstawiony na Rys. 6.13. Obiekt ten jest liniowy, asymptotycznie stabilny i posiada dwie inercje. Dokładne stałe czasowe oraz wzmocnienie, które opisują ten obiekt nie będą podawane, gdyż inaczej zadanie identyfikacji (tj. pozyskiwanie modelu w postaci odpowiedzi skokowej) miałyby się z celem (znając równania obiektu można by taką odpowiedź wygenerować).

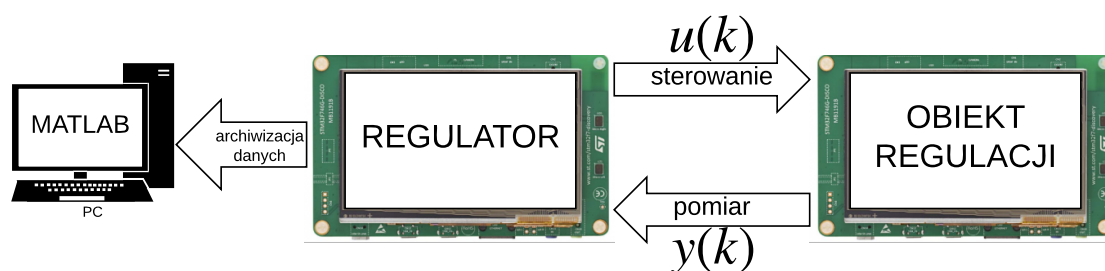
Sterowanie obiektem odbywa się poprzez zmianę wartości napięcia (w zakresie od 0 do 3,3 V) na jednym z jego pinów. Wyjściem obiektu jest również wartość napięcia (także w zakresie od 0 do 3,3 V), które można zmierzyć na jednym z jego pinów. Na wyświetlaczu obiektu rysowane są na bieżąco wartości sygnału wyjściowego obiektu (**Y1**) oraz sygnału wejściowego obiektu, tj. sterowania (**U1**). 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 płytek STM32F746G-DISCOVERY) realizowane jest przez prowadzącego.

6.2.7. 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 płytke wyposażoną w złącza zgodne ze złączem Arduino Uno.

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

```
1 void updateControlSignalValue(uint16_t out){
2 static uint8_t buffertx[] = {0x0F, 0xFF};
3 buffertx[0] = (out>>8)&0xFF;
4 buffertx[1] = (out>>0)&0xFF;
5 HAL_I2C_Master_Transmit(&hi2c1, 0xC2, (uint8_t*)buffertx, 2,1000);
6 }
```



Rys. 6.13. Schemat połączenia obiektu symulowanego i regulatora

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 znikomy czas wykonania są doskonałymi argumentami, aby dopuścić się takiego zabiegu.

6.2.8. 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); % zamknięcie wszystkich połączeń 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 kanału komunikacyjnego

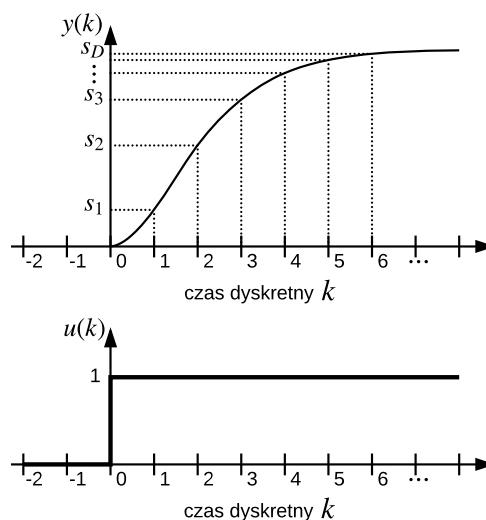
y(1:100) = -1; % mikrokontroler zwraca tylko dodatnie wartości, więc każde -1
% oznacza, że nie otrzymaliśmy jeszcze wartości o tym indeksie
while true
    txt = fread(s,16); % odczytanie z portu szeregowego
    eval(char(txt)); % wykonajmy to co otrzymaliśmy
    if(y(1) ~= -1)
        break;
    end
end
y(2:100) = -1; % ignorujemy wszystko co odczytaliśmy poza pierwszym elementem
while true
    txt = fread(s,16); % odczytanie z portu szeregowego
    eval(char(txt)); % wykonajmy to co otrzymaliśmy
    if(min(y) ~= -1) % jeśli najmniejszym elementem nie jest -1, to znaczy że
        break; % nie ma brakujących elementów, a więc dane są 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 " $y(\%3d) = \%4d;\backslash n\backslash r$ ", 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 odczekaniu 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, pojawia 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 sprzed skoku wartości sterowania. Zależność mię-



Rys. 6.14. Schemat zależności czasowych między zmianą sygnału sterowania a pomiarem wyjścia obiektu regulowanego

dzy czasem aplikacji nowej wartości sterowania a chwilą pomiaru wyjścia obiektu regulacji przedstawia Rys. 6.14.

Skalowanie pomiarów tak, aby uzyskać odpowiedź skokową polega na odjęciu od wszystkich pomiarów wartości pomiaru wyjścia obiektu regulowanego sprzed zmiany 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; % przeskalowane pomiary = jednostkowa odpowiedź 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.6.14). 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 obliczeń 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 skokową 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).

6.2.9. 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 by 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 kanału 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.

6.2.10. Zadanie do wykonania

Zadaniem, które będzie podlegało późniejszej ocenie jest implementacja na mikrokontrolerze rodziny 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 możliwie dużym horyzoncie 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 zakłócenia (realizowanego poprzez wciśnięcie 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.

6.3. Wykonanie 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.

7. Projekt 2: Identyfikacja modeli (typu odpowiedzi skokowej) procesu laboratoryjnego, implementacja algorytmu regulacji DMC, dobór nastaw algorytmu, badania porównawcze

7.1. Wprowadzenie

Celem tego projektu jest implementacja algorytmu regulacji DMC z użyciem obiektu rzeczywistego (tj. takiego, którego sygnały mają pewną reprezentację fizyczną). Wykorzystane zostanie stanowisko grzejąco-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.

7.2. Treść projektu

7.2.1. Stanowisko grzejąco-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 przysyłać 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 przysyła wartości temperatury ze skończoną 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.

7.2.2. Stany awaryjne

Rozważane stanowisko posiada dwa kluczowe stany awaryjne, które mogą zostać w łatwy sposób wykryte i jednoznacznie zidentyfikowane. Te dwa aspekty należy wyraźnie

rozdzielić, 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 zauważyć, 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 mogłaby być ostatnia z wymienionych sytuacji nieprzyjemnych – tj. błąd wykonania funkcji – lecz wspomniane stanowisko grzejąco-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 poszukiwaniami 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 wprowadzone 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.

7.2.3. Wizualizacja

W poprzednim projekcie płytka odpowiadająca za symulację procesu regulacji jednocześnie wizualizowała jego stan. W tym projekcie wizualizacja ma mieć miejsce na płycie 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 szczegóły 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 znajduje 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.

7.2.4. 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.

7.2.5. 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.

7.3. Wykonanie projektu

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 wyświetlanego obrazu lub wykorzystanie przerwania `HAL_LTDC_LineEvenCallback` do modyfikacji wyświetlanego obrazu w czasie powrotu plamki,
- Obsługę ekranu dotykowego z użyciem okresowego odpytywania (szczegóły 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.

8. Projekt 3: System operacyjny czasu rzeczywistego

8.1. Wprowadzenie

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.

8.2. Treść projektu

8.2.1. 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 definicję 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 osiąganе”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 oczekiwanego czasu odpowiedzi. Jako przykład można

¹ P. 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.

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.

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 współbieżnie (nie mylić z wykonywaniem równoległym, które wymaga istnienia wielu procesorów). Takie podejście pozwala na jednoczesne 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 grzejąco-chłodzące), to zaplanowanie wykorzystania przerwań generowanych przez poszczególne timery jest zadaniem co najmniej niełatwym (należy pamiętać, że timery 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.

8.2.2. 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.

8.2. TREŚĆ PROJEKTU

Przed 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:

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

gdzie typ `task_table_t` zdefiniowany jest jako

```
1 typedef struct{ // structure of task_table
2     uint32_t sp; // Task stack pointer
3     int flags; // Task status flags
4 } 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

```
1 task_table[1].sp = (unsigned int) 0x20001700;
2 task_table[2].sp = (unsigned int) 0x20002700;
3 new_task(my_task_1, (uint32_t) &var1); // inicjalizuje Task1
4 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` (zgłaszane 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:

```
1 void my_task_1(void *data){
2     float angle = 0.0f;
3     float l = 50.0f;
4
5     unsigned int varTask;
6
7     varTask = *((unsigned int*)data);
8     while(1){
9         ++counter_task1;
10        angle += dir1*0.001f;
11        animation(angle,l);
12    }
13 }
```

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 rzutować 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 współdzielonego zasobu. W szczególności zapisy dokonywane są przez poszczególne taski do osobnych fragmentów pamięci, a peryferiale obsługiwane są osobno, każdy w dokładnie jednej funkcji obsługi przerwania.

Zaimplementowane są trzy kluczowe funkcje obsługi przerwania (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 wciśnięta zostaje jedna z połówek ekranu dotykowego, póty 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 – Hard-Fault. 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 wyświetlone 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.

8.2.3. Zadanie do wykonania

Student ma wykonać następujące zadania:

- Poprawić obecną implementację, tak, aby kwadraty obracały się wyłącznie na dotknięcie ekranu:
 - lewa górna część – lewy kwadrat obraca się zgodnie z ruchem wskazówek zegara,
 - lewa dolna część – lewy kwadrat obraca się przeciwnie do ruchu wskazówek zegara,
 - prawa górna 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.

8.3. Wykonanie projektu

Zadaniem jest implementacja kilku funkcjonalności i wykorzystanie kilku mechanizmów znanych z systemów operacyjnych. Funkcjonalności to:

- Task Manager

Jest to bardzo uproszczony menadżer zadań. Jedyne co musi potrafić to zliczać ile razy rozpoczął się timer/rozpoczęła się nowa pętla taska w ciągu ostatniej **sekundy**. Wartości te powinny być pokazane w postaci **słupków na wyświetlaczu**.

- Regulatory typu PID z przycinaniem sterowania

Jeden regulator trzeba zaimplementować od zera a drugi poprawić.

Pierwszy regulator już jest zaimplementowany. Jest to regulator który służy do regulacji w jednej z osi procesu regulacji, który należy interpretować jako kulka na płaszczyźnie, której pochyłem Państwo sterują. Mimo, że ten regulator działa zgodnie z intuicją, to jest on zaimplementowany niepoprawnie (z punktu widzenia wykorzystania adekwatnego mechanizmu systemu operacyjnego). Stąd też **potrzeba aby go Państwo poprawili**. Zaznaczam, że algorytm PID jest poprawny – problem jest w jego umiejscowieniu w systemie operacyjnym. Okres próbkowania powinien pozostać równy **200ms**.

Drugi regulator należy zaimplementować od zera – mogą Państwo czerpać garściami z doświadczenia z pierwszym regulatorem z szablonu. Oczywiście należy ten regulator zaadaptować w taki sposób, aby regulował kulkę w osi prostopadłej do pierwszego regulatora. Dodatkowo chciałbym aby ten regulator pracował z 5-krotnie mniejszym okresem próbkowania (tj. pierwszy działa z okresem 200ms, natomiast drugi z okresem **40ms**).

- Przełączanie parametrów regulatorów

Chciałbym aby Państwo zaimplementowali na wyświetlaczu i ekranie dotykowym **przycisk służący do przełączania parametrów jednego z regulatorów**. Zmiana parametrów nie musi być znaczna, a w szczególności nowe parametry nie muszą być bardzo dobrze dobrane – ważne, żeby zmiana była wyraźnie widoczna w działaniu regulatora (np. wyłączyć całkę, wyłączyć różniczkę i zrobić duże wzmocnienie). Chciałbym, aby Państwo to zrealizowali **poprzez zastosowanie kolejki** do przesłania regulatorowi **trzech** wiadomości – każda z informacją o nowym parametrze regulatora PID. Wiem, że da się to zrobić w ramach jednej struktury. Ważne jest, aby **kolejność przesłanych wiadomości była dowolna** z punktu widzenia regulatora.

- Statyczna animacja

Ostatnie zadanie to implementacja **statycznej animacji w zwykłym Tasku**. Niech to będzie obracający się kwadrat z kołem na jednym rogu, aby było łatwo liczyć jego obroty. Prędkość obrotowa kwadratu niech wynosi 0,2 obrotu na sekundę. Ważne jest, aby prędkość obrotów była zawsze ta sama bez względu na to jakie Państwo wstawiają do tego Tasku `osDelay`! Polecam zapoznać się z takim bytem jak `osKernelGetTickCount()`

– funkcja ta zwraca liczbę milisekund, która minęła od (z grubsza) uruchomienia mikrokontrolera.

8.3.1. Ocenianie

Komplet punktów uzyskają Państwo jeśli zaimplementują/poprawią Państwo powyższe funkcjonalności i przy okazji nie doprowadzą do mrygania wyświetlacza (tj. błędów wyświetlania).

Oczywiście na koniec należy zaprezentować działający projekt i pełnię funkcjonalności. Mają Państwo pełną dowolność w kwestii modyfikacji kodu, aczkolwiek zachęcam aby nie ruszali Państwo funkcji obsługi przerwań. Sterowniki natomiast warto aby Państwo modyfikowali, gdyż jest to potrzebne do dodania nowych grafik i obsługi ekranu dotykowego. Oczywiście szerszy opis kodu, który będą Państwo używać w ramach tego projektu znajdą Państwo w ostatnio nagrany zdalnym „wykładzie” gdzie omawiam przykładowy kod programu we FreeRTOS. Szczerze zachęcam do zapoznania się z nim, gdyż mogą Państwo znaleźć tam odpowiedzi na niektóre Państwa pytania i nie tylko takie odpowiedzi.

Podkreślam: Z tego projektu nie generuję Państwo sprawozdań. Wszystko jest oceniane na podstawie prezentacji (jak laboratorium 2, 3 i 4.)

8.3.2. Uwagi

— Rozszerzone implementacje funkcji z BSP do wyświetlacza

Na potrzeby podwójnego buforowania wygenerowałem odpowiedniki klasycznych funkcji z BSP do rysowania grafik na wyświetlaczu. Funkcje te kończą się na `_AtAddr` i przyjmują dodatkowy argument (ostatni w liście) będący adresem bufora do którego Państwo chcą zapisać zmiany w postaci grafiki. Wynika to z faktu, iż przy podwójnym buforowaniu nie możemy pisać po obecnie „wybranym” przez wyświetlacz buforze, bo to by generowało kłopoty wyświetlania. Dlatego też każda operacja rysowania musi być wykonana na buforze wyłącznie do zapisu. Ponieważ jednak mamy dwie warstwy i obie podwójnie buforujemy, to mamy też dwie opcje co do tego co wpisać jako ostatni atrybut: `layer[0]` – spodnia warstwa, i `layer[1]` – wierzchnia warstwa. Obecnie w kodzie żadna nie ma dedykowanego przeznaczenia, więc jeśli wszystkie grafiki postanowią Państwo umieścić Państwo na jednej warstwie, to też będzie dobrze. Każda warstwa jest czyszczona do zera na początku funkcji sterownika, więc nie ma obaw o artefakty.

— Opis szablonu projektu

W programie szablonowym mają Państwo zaimplementowane funkcje obsługi przerwań:

`HAL_GPIO_EXTI_Callback` – obsługiwany jest tutaj ekran dotykowy `HAL_LTDC_LineEventCallback` – obsługa podmiany buforów wyświetlacza

Funkcji powyższych proszę nie zmieniać. Dodatkowo zaimplementowane są taski:

- `ltdcDriver` – obsługa wyświetlacza, tutaj Państwo wstawiają kod do rysowania
- `processDriver` – obsługa procesu regulacji – proszę nie tykać
- `tsDriver` – obsługa ekranu dotykowego, należy rozwinąć tutaj funkcjonalność zgodnie z treścią zadań
- `controllerV` – regulator pozycji w osi pionowej – do naprawy. (są z grubsza dwa podejścia do tego problemu – proszę kombinować)
- `staticAnimation` – tutaj proszę zaimplementować rysowanie statycznej animacji (proszę testować z różnymi wartościami `osDelay()`)

Poza tym są też stworzone pewne kolejki:

- `qProcess` – do komunikacji między `processDriver` a regulatorami
- `qLTDCReady` – do komunikacji między `ltdcDriver` a `HAL_LTDC_LineEventCallback`
- `TSSState` – do komunikacji między `tsDriver` a `HAL_GPIO_EXTI_Callback`
- `qControllerV` – do komunikacji między `processDriver` a regulatorem `controllerV`
 - tutaj przychodzi pomiar sygnału wyjściowego procesu

Ostatecznie występuje też jeden timer o nazwie `process`, który symuluje rzeczywisty proces z którym komunikujemy się np. przy użyciu protokołu MODBUS. Jest także semafor `sLTDC`, wykorzystywany do zabezpieczania buforów przed jednoczesnym zapisem i odczytem ze strony sterownika wyświetlacza i przerwania z nim związanego. To ostatnie jest także synchronizowane z wykorzystaniem zdarzenia `eLTDCReady`.

Dodatkowo w kodzie występują komentarze zaczynające się od **TODO**, **FIXME** albo **LOOKATME**. Gdzie wskazałem Państwu pewne ważne/ciekawe miejsca w kodzie. Nie jest ich dużo, ale wydaje mi się że są kluczowe, szczególnie, że zazwyczaj są to miejsca podlegające modyfikacji lub rozszerzeniu funkcjonalności. Pozostałe funkcje tasków/timerów obsługę kolejek piszą Państwo sami (z pomocą automatycznej generacji kodu).