

Wydział Elektroniki i Technik
Informacyjnych
Politechnika Warszawska

Systemy Mikroprocesorowe w Sterowaniu
Ćwiczenia laboratoryjne

Patryk Chaber, Andrzej Wojtulewicz

Warszawa 2022

Spis treści

1. Ćwiczenie 1: Podstawy pracy z zestawem uruchomieniowym	3
1.1. Wprowadzenie	3
1.2. Treść ćwiczenia	3
1.2.1. Tworzenie pierwszego projektu w STM32CubeIDE	3
1.2.2. STM32CubeIDE	3
1.2.3. Podłączenie mikrokontrolera	4
1.2.4. Stworzenie projektu	7
1.2.5. Wgranie programu na mikrokontroler	14
1.2.6. Konfiguracja sprzętowa mikrokontrolera	21
1.2.7. Konfiguracja zegarów mikrokontrolera	22
1.2.8. Odczyt wartości cyfrowej	25
1.2.9. Obsługa alfanumerycznego wyświetlacza LCD	26
1.2.10. Konfiguracja PWM	28
1.2.11. Odczyt i wykorzystanie wejścia analogowego	30
1.3. Wykonanie ćwiczenia	32
2. Ćwiczenie 2: obsługa prostych czujników i urządzeń wykonawczych mikroprocesorowego systemu automatyki przy wykorzystaniu systemu przerw	34
2.1. Wprowadzenie	34
2.2. Treść ćwiczenia	34
2.2.1. Wykorzystane mechanizmy	34
2.2.2. Przebieg laboratorium	40
2.3. Wykonanie ćwiczenia	52
3. Ćwiczenie 3: Obsługa złożonych czujników i urządzeń wykonawczych mikroprocesorowego systemu automatyki	53
3.1. Wprowadzenie	53
3.2. Treść ćwiczenia	53
3.2.1. Pętla prądowa 4-20 mA	53
3.2.2. Cyfrowy przetwornik temperatury z termometrem rezystancyjnym	54
3.2.3. Transmisja szeregową	56
3.2.4. Implementacja na ZL27ARM	57
3.2.5. Transmisja szeregową – standard MODBUS RTU	59
3.2.6. Stanowisko grzejąco-chłodzące	63
3.3. Wykonanie ćwiczenia	65

1. Ćwiczenie 1: Podstawy pracy z zestawem uruchomieniowym

1.1. 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.

1.2. Treść ćwiczenia

1.2.1. 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.

1.2.2. STM32CubeIDE

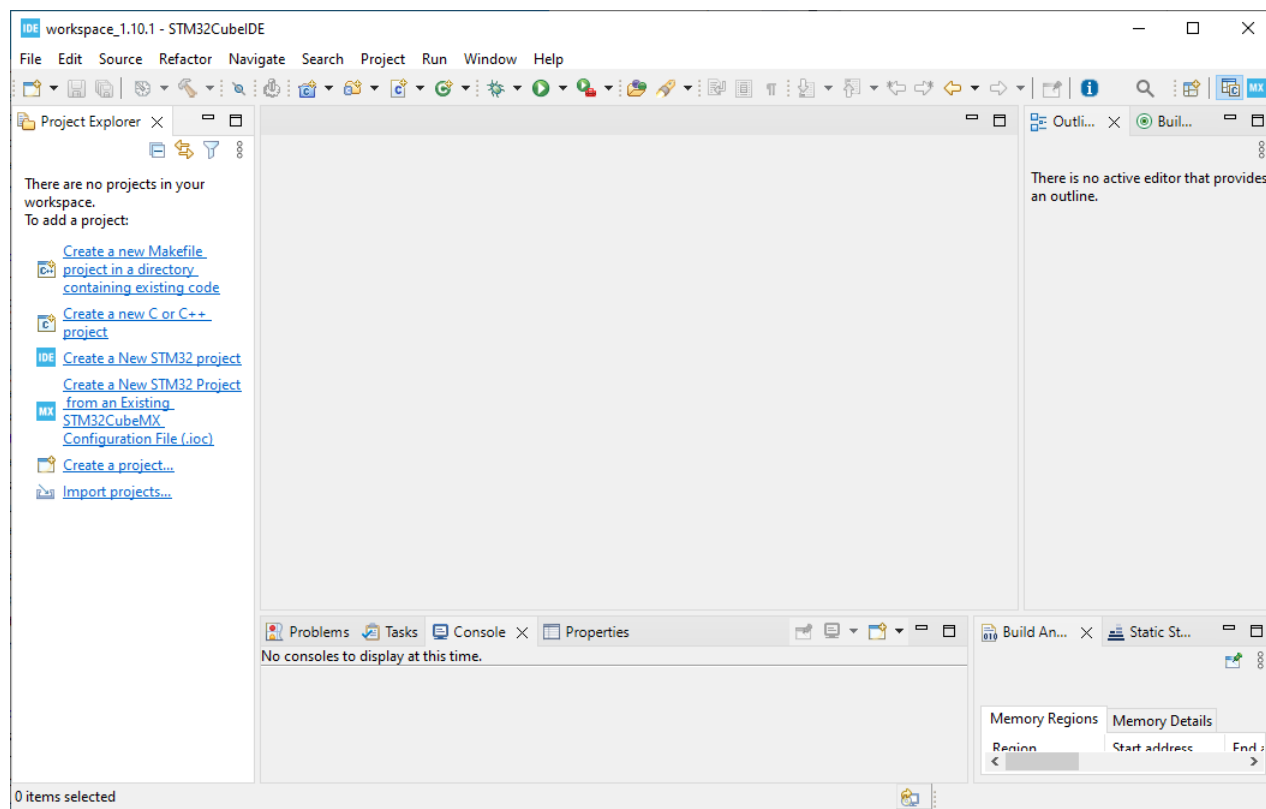
Zintegrowane środowisko deweloperskie o nazwie STM32CubeIDE bazuje na oprogramowaniu Eclipse IDE. Powoduje to, że często w trakcie korzystania z tego oprogramowania może pojawiać się odniesienie do nomenklatury związanej z tym środowiskiem, takie jak „perspektywa” czy „katalog roboczy”. Perspektywa, 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. 1.1. 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*).

1.2.3. Podłączenie mikrokontrolera

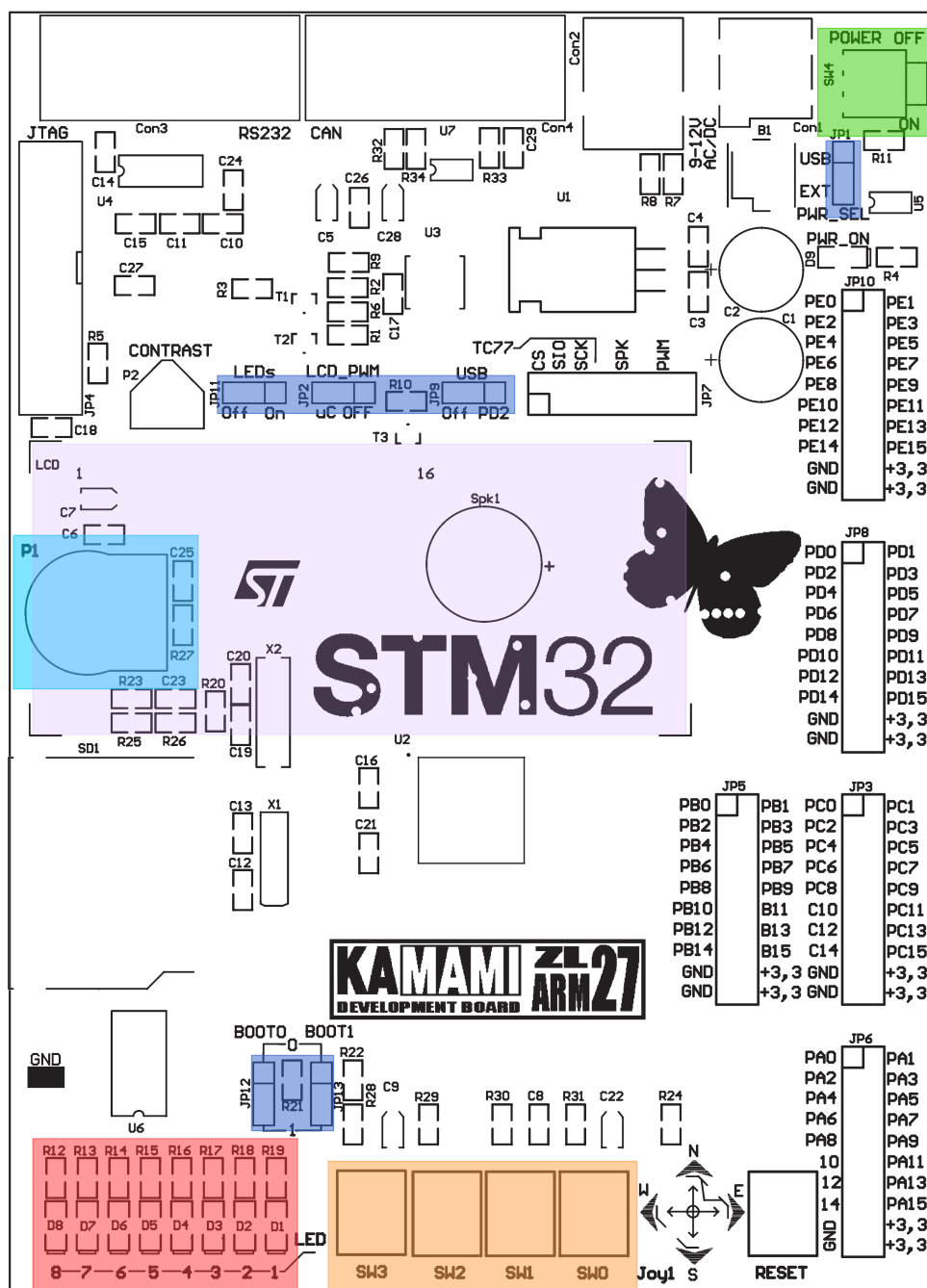
Jako mikrokontroler użyty zostanie STM32F103VBT6, zawierający się w zestawie uruchomieniowym ZL27ARM (rys. 1.2). Ogólne informacje dotyczące tego zestawu znajdują się w Instrukcji Użytkownika związanej z tym zestawem (tam też znajduje się schemat płytki, który jest widoczny na rys. 1.3), natomiast szczegółowe 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. 1.4. Jest on podłączany poprzez złącze *JTAG* (*Joint Test Action Group*), które pozwala na testowanie (w tym debugowanie



Rys. 1.1. Okno środowiska STM32CubeIDE



Rys. 1.2. Płytką rozwojowa ZL27ARM z mikrokontrolerem STM32F103VBT6



Rys. 1.3. 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.

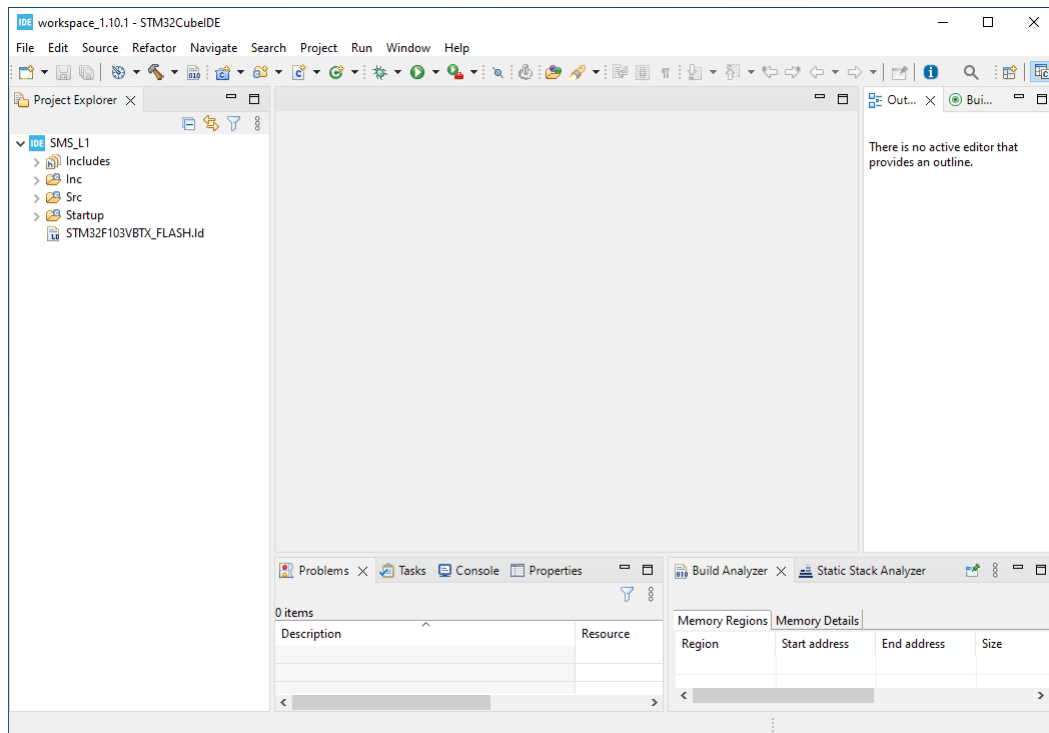
1.2.4. Stworzenie projektu

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. 1.6). Należy wybrać odpowiedni i wcisnąć przycisk *Next*. W kolejnym oknie (rys. 1.7) 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. 1.5.

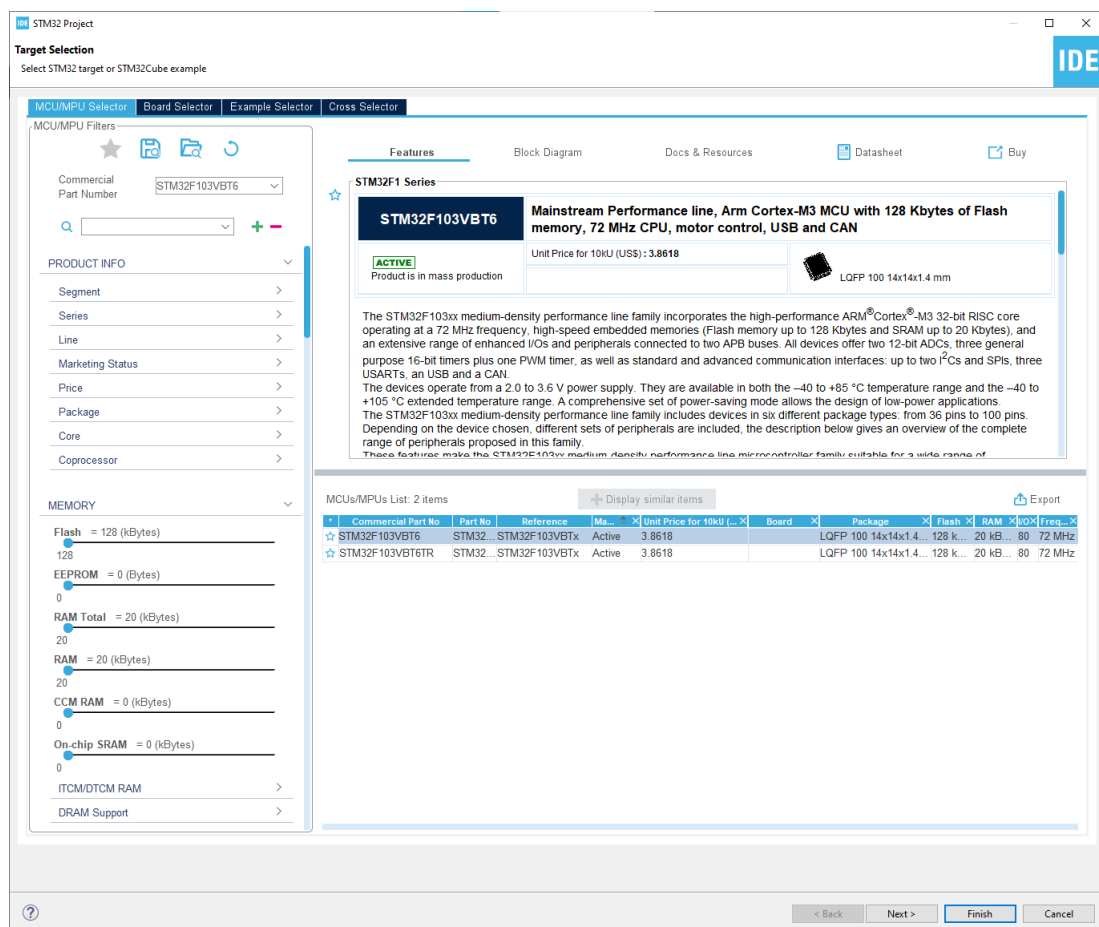
Obecnie przygotowany kod pozwala na tworzenie projektów z wykorzystaniem pisania po rejestrach. Jest to metoda pisania oprogramowania na mikrokontrolery rekomendowana przez wielu programistów, jako, że zapewnia niemal absolutną kontrolę nad wykonaniem programu. W ramach tych ćwiczeń nie będziemy skupiali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



Rys. 1.4. Programator firmy SEGGER, *J-LINK EDU*

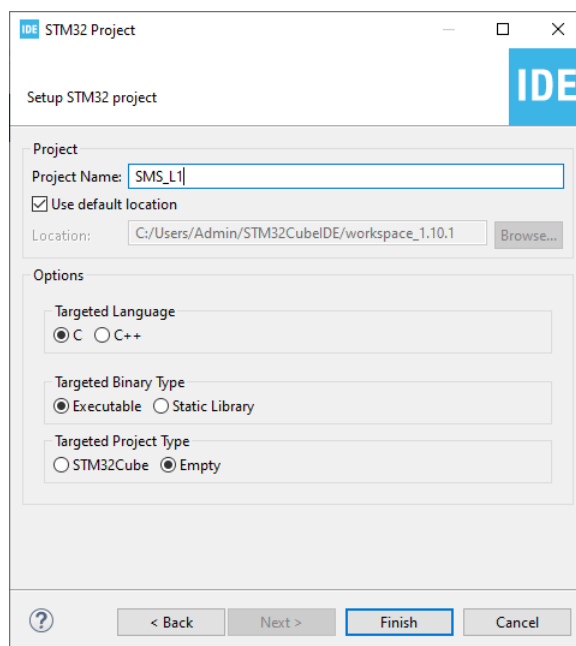


Rys. 1.5. Okno środowiska STM32CubeIDE bezpośrednio po stworzeniu pierwszego projektu

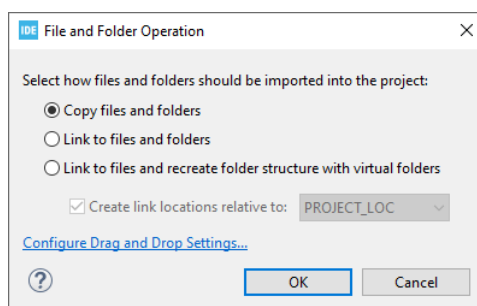


Rys. 1.6. Okno wyboru platformy docelowej (Target Selection)

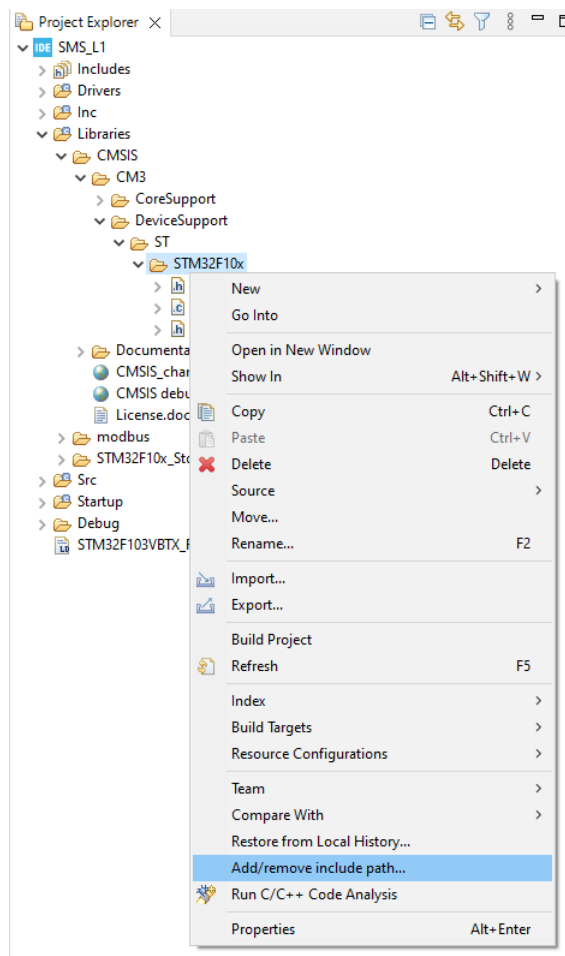
1.2. TREŚĆ ĆWICZENIA



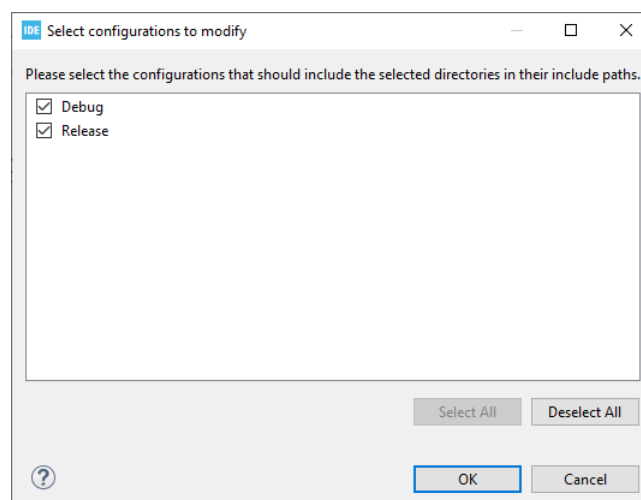
Rys. 1.7. Okno podstawowej konfiguracji projektu



Rys. 1.8. Okno wyboru sposobu importowania katalogów z plikami do projektu



Rys. 1.9. Menu kontekstowe pozwalające na dodanie katalogu projektu do listy ścieżek z plikami nagłówkowymi



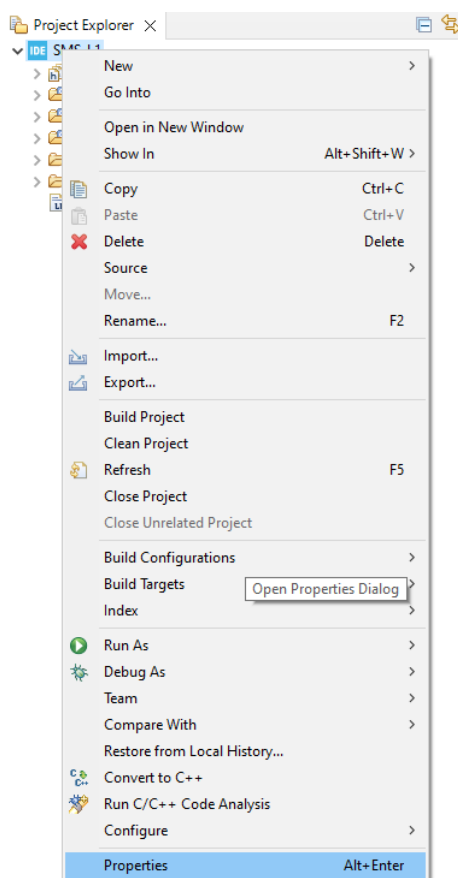
Rys. 1.10. Okno wyboru konfiguracji dla której pliki nagłówkowe mają być dodane

1.2. TREŚĆ ĆWICZENIA

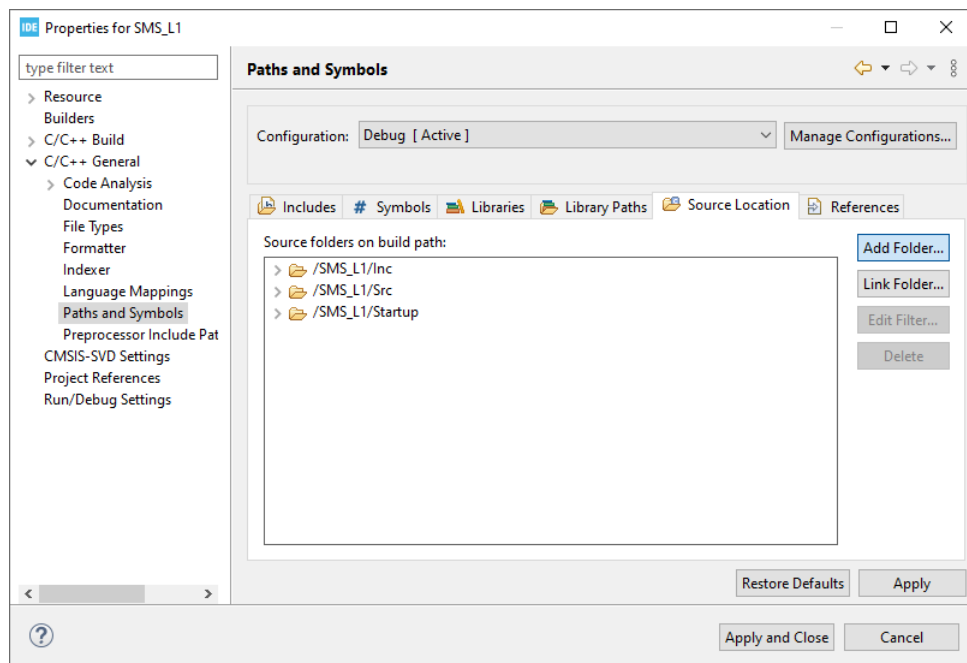
Korzystanie z bibliotek SPL wymaga kilku dodatkowych zabiegów. Jako pierwsze należy dodać bibliotekę SPL i przy okazji warto wyposażyć się w podstawowe sterowniki, m.in. do wyświetlacza LCD1602. W tym celu z katalogu wskazanego przez prowadzącego należy przeciągnąć metodą „przeciągnij i upuść” katalogi o nazwie *Drivers* oraz *Libraries* na korzeń projektu, aby trafiły do głównego katalogu projektu. Po przeciągnięciu ukazuje się okno (rys. 1.8) 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. 1.9), a w oknie, które się pojawiło (rys. 1.10) 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

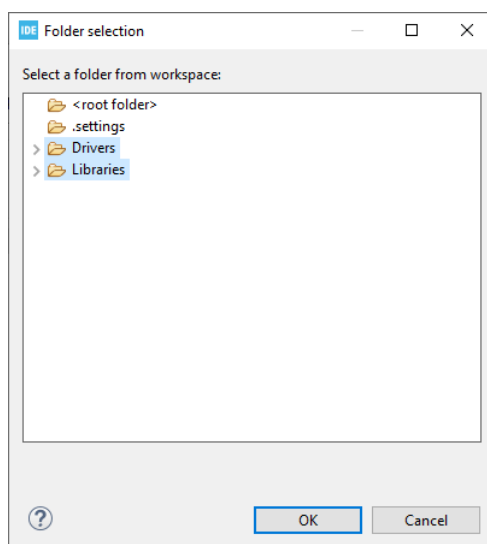
Następnie należy wskazać kompilatorowi, że w dodanych katalogach znajdują się pliki, które powinny zostać skompilowane. W tym celu należy kliknąć prawym przyciskiem my-



Rys. 1.11. Menu kontekstowe projektu pozwalające na otwarcie szczegółowych ustawień projektu



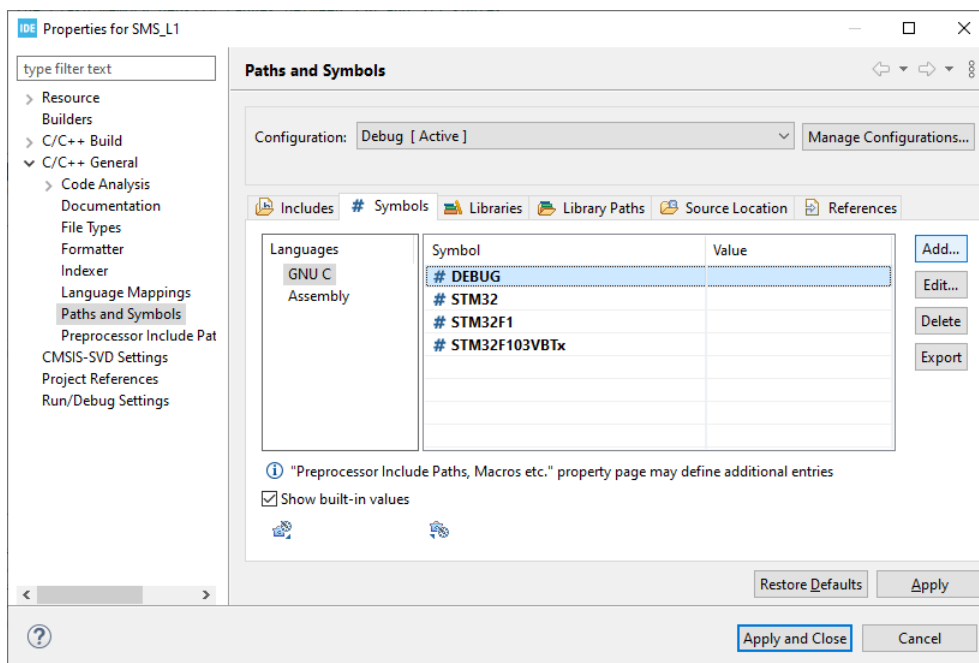
Rys. 1.12. Okno szczegółowych ustawień projektu – ścieżki z plikami źródłowymi



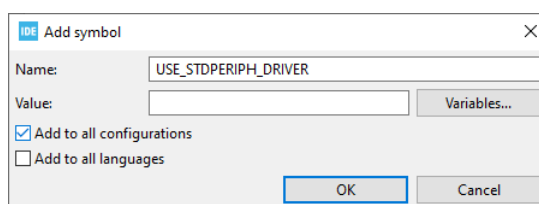
Rys. 1.13. Okno pozwalające na wybór katalogów z plikami źródłowymi

1.2. TREŚĆ ĆWICZENIA

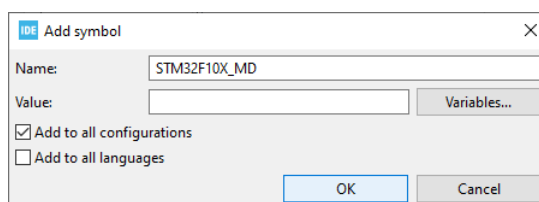
szy na korzeń projektu (rys. 1.11) i wybrać *Properties*. W oknie, które się pojawi, w lewej jego części, należy wybrać *C/C++ General* → *Paths and Symbols*. W centralnej/prawej części okna aktywuje się zakładka *Includes*, gdzie zobaczyć można będzie, przed chwilą dodane, ścieżki zawierające pliki 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. 1.12). Tutaj należy kliknąć przycisk *Add Folder...* a następnie wybrać katalogi *Libraries* oraz *Drivers* (rys. 1.13) i wcisnąć przycisk OK. Kompilator przeszuka nie tylko te konkretne katalogi, ale także wszystkie zagnieżdżone katalogi w tych katalogach.



Rys. 1.14. Okno szczegółowych ustawień projektu – symbole preprocesora



Rys. 1.15. Okno dodawania nowego symbolu preprocesora – symbol informujący kompilator o wykorzystaniu bibliotek SPL



Rys. 1.16. Okno dodawania nowego symbolu preprocesora – symbol informujący kompilator o wykorzystaniu instrukcji dla mikrokontrolera o „średniej gęstości” (*Medium Density*)

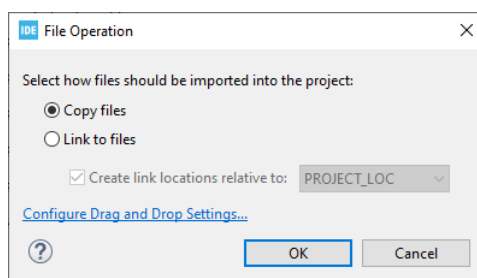
Należy także dodać dwa symbole preprocesora wykorzystywane w plikach nagłówkowych biblioteki SPL. W tym celu należy przejść do zakładki *Symbols* (rys. 1.14) i po kliknięciu przycisku *Add...* wpisać w pole *Name:* wyrażenie `USE_STDPERIPH_DRIVER`, zaznaczyć *Add to all configurations* (rys. 1.15), po 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. 1.16), 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. 1.17, gdzie należy wybrać kopiowanie (pierwszą opcję) i zaakceptować to przyciskiem *OK*. Drzewo projektu na tym etapie powinno wyglądać jak na rys. 1.18. W tym momencie program powinien się skompilować z powodzeniem, co w widoku konsoli powinno wyglądać podobnie jak na rys. 1.19. W przyszłości może wystąpić sytuacja, w której użytkownik rozpocznie kompilację bez zapisania uprzedniego wszystkich zmodyfikowanych plików – pojawi się wtedy okno na rys. 1.20, 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. 1.11), 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. 1.21).

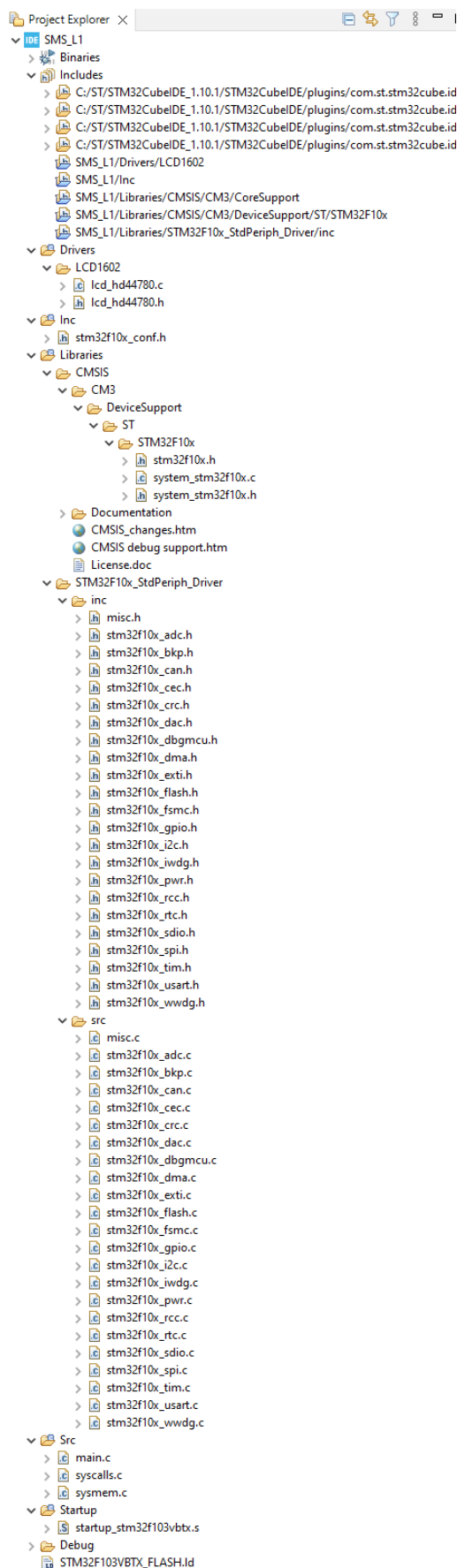
1.2.5. Wgranie programu na mikrokontroler

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. 1.22. W tym oknie zdefiniowana jest nazwa konfiguracji wykorzystanej do wgrywania oprogramowania na mikrokontroler. Interesującą nas zakładką jednak będzie zakładka *Debugger* (rys. 1.24), gdzie należy zmienić programator na odpowiedni. W tym celu należy w polu *Debug pro-*

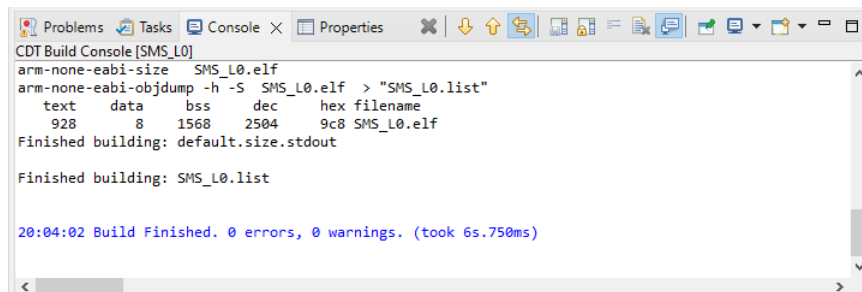


Rys. 1.17. Okno wyboru sposobu importowania plików do projektu

1.2. TREŚĆ ĆWICZENIA



Rys. 1.18. Widok drzewa projektu gotowego do implementacji pierwszego programu z wykorzystaniem biblioteki SPL



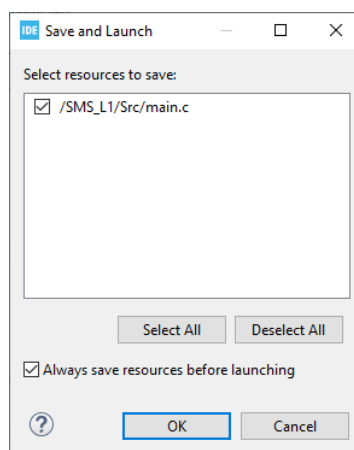
CDT Build Console [SMS_L0]

```
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

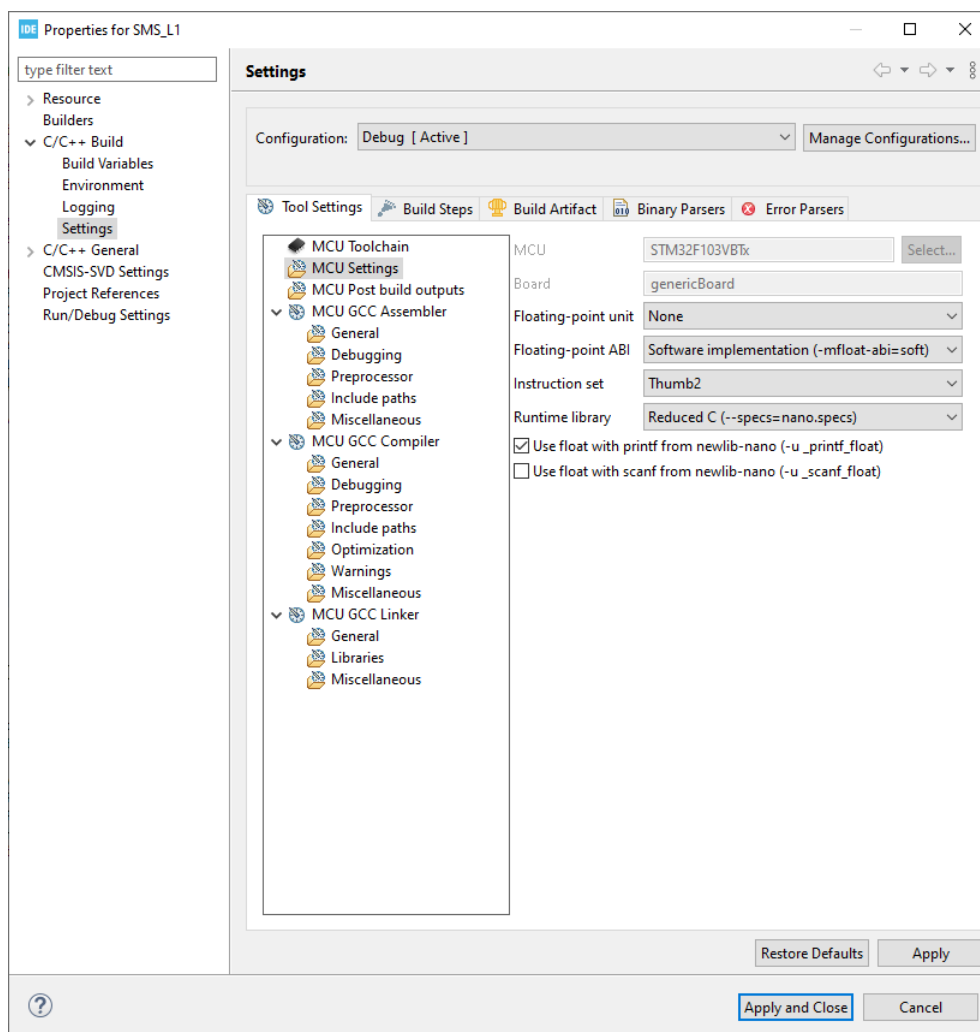
20:04:02 Build Finished. 0 errors, 0 warnings. (took 6s.750ms)
```

Rys. 1.19. Widok konsoli informujący o poprawnym przebiegu kompilacji

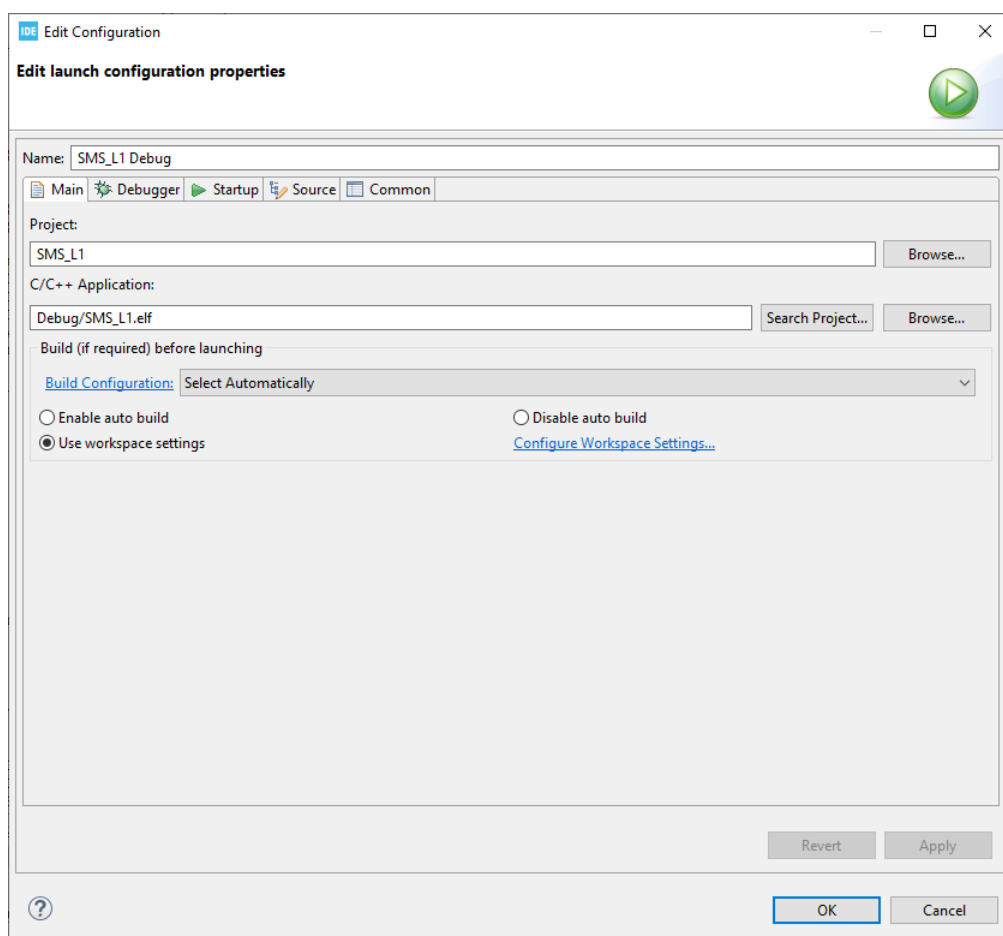


Rys. 1.20. Okno informujące o istnieniu niezapisanych zmian w momencie rozpoczęcia kompilacji

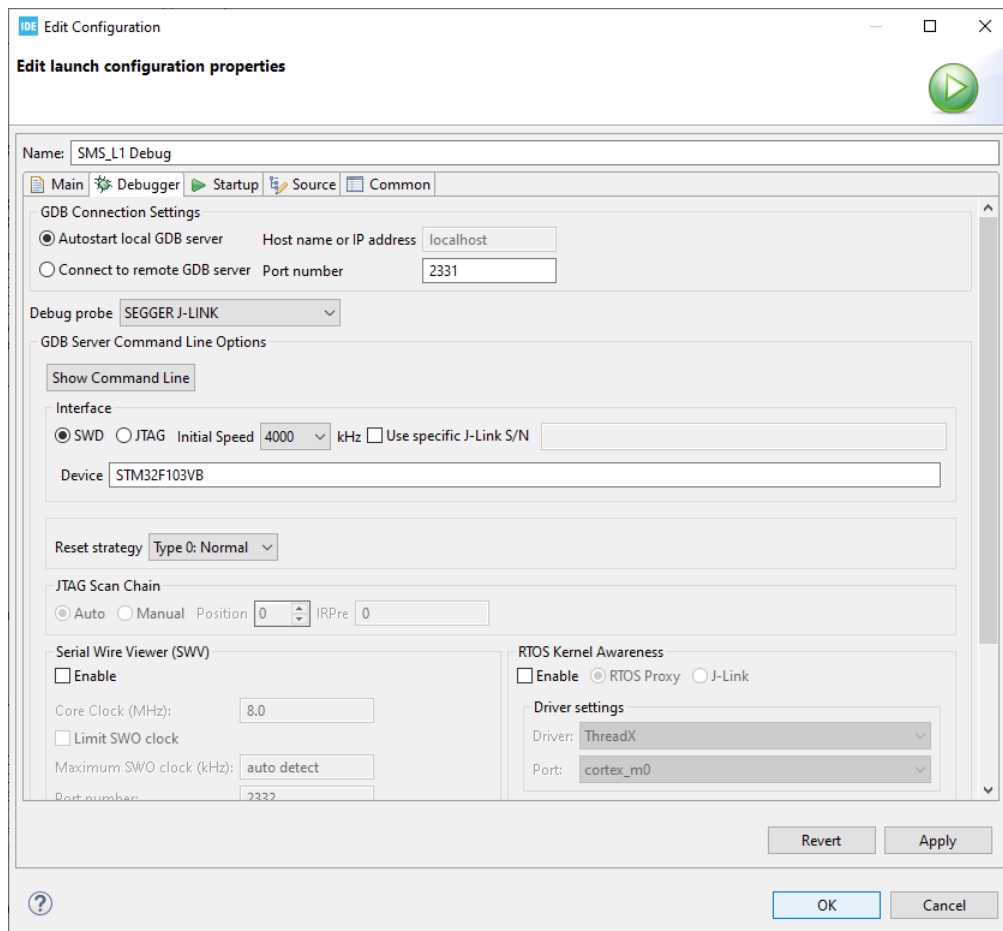
1.2. TREŚĆ ĆWICZENIA



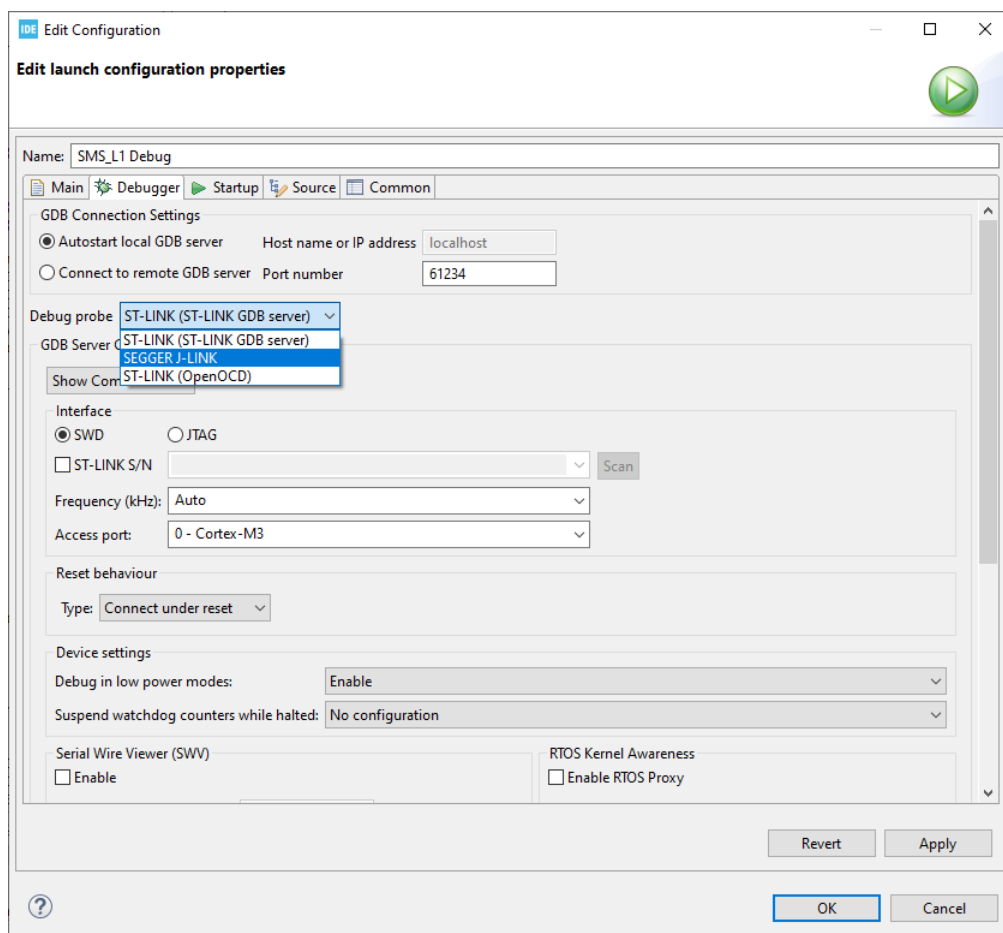
Rys. 1.21. Okno szczegółowych ustawień projektu – przydatne ustawienia kompilacji



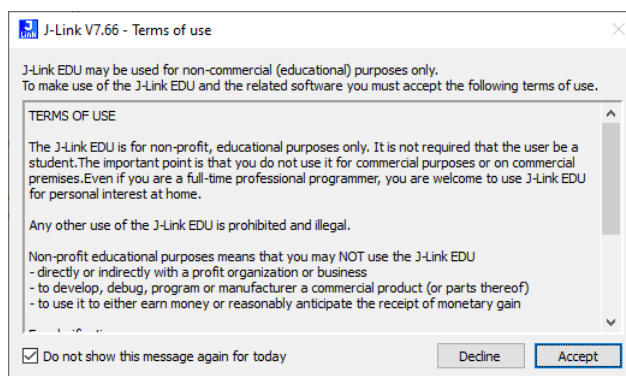
Rys. 1.22. Okno konfiguracji sposobu wgrywania programu na mikrokontroler – zakładka główna



Rys. 1.23. Okno konfiguracji sposobu wgrywania programu na mikrokontroler – zakładka konfiguracji debuggera z domyślną konfiguracją debuggera SEGGERJ-LINK



Rys. 1.24. Okno konfiguracji sposobu wgrzywania programu na mikrokontroler – zakładka konfiguracji debuggera



Rys. 1.25. 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. 1.4. Po wybraniu odpowiedniego programatora powinno ukazać się okno z dodatkowymi opcjami jak na rys. 1.23 – 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. 1.25). 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.

1.2.6. Konfiguracja sprzętowa mikrokontrolera

Istotne, na potrzeby tego i kolejnych ćwiczeń, elementy zestawu uruchomieniowego ZL27ARM zostały zaznaczone na rys. 1.3. 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 zwerek 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*.

1.2.7. Konfiguracja zegarów mikrokontrolera

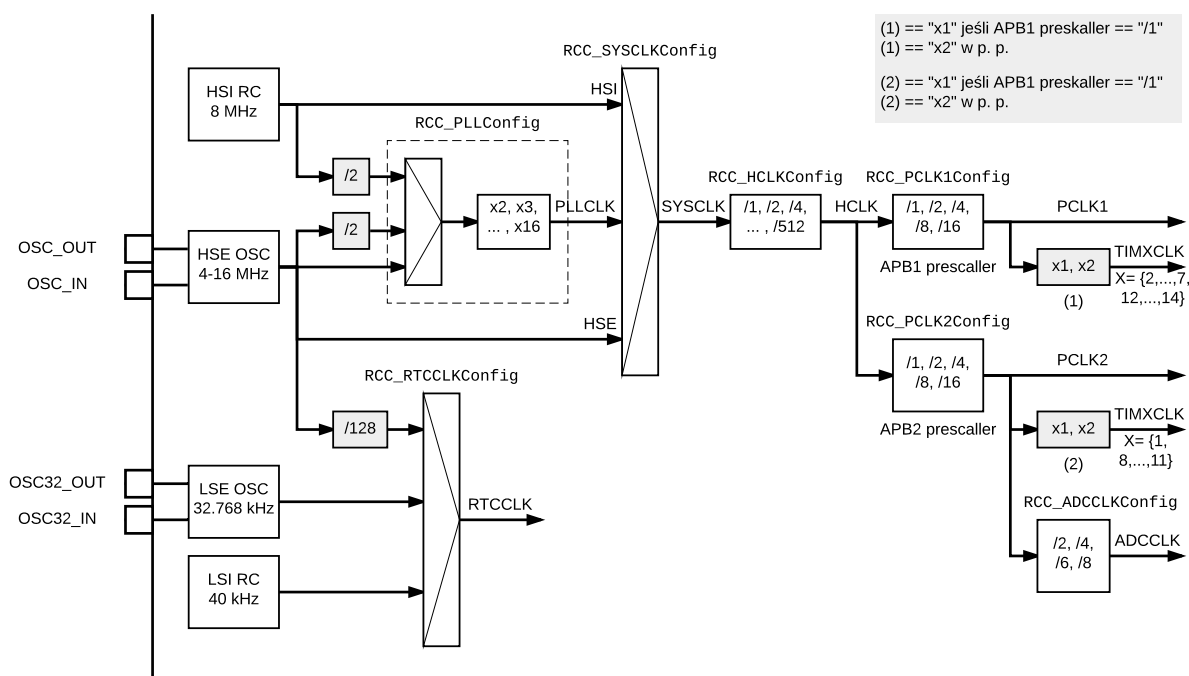
Mikrokontroler składa się z wielu (tysięcy) bramek logicznych, które pracują w trybie synchronicznym. Oznacza to, że są one taktowane zegarem i wraz z nim zmieniają wartość na wyjściu bramek. Dzięki temu unika się takich problemów jak zjawisko hazardu lub wyścigu, a więc problemów wynikających z niezerowego 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. 1.26. Na schemacie są dodatkowo umieszczone nazwy funkcji ze stan-



Rys. 1.26. Uproszczony schemat układu taktowania procesora i peryferali – pełna wersja znajduje się w dokumencie RM0008 (Rys. 8)

dardowej biblioteki do obsługi peryferiali, 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. 1.26), 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 1.2.7, 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 jakiejkolwiek interakcji ze strony użytkownika. Jest to ciekawe zjawisko wynikające z niepodciągnięcia wyjść prowadzących do diod, które niestety nie zostanie tutaj szczegółowo omówione. Należy jednak pamiętać, że zjawisko to ma wpływ wyłącznie na piny, które nie są skonfigurowane jako wyjścia – na tę chwilę nie należy się tym przejmować.

Poniżej została przedstawiona przykładowa funkcja konfiguracyjna 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. 1.26).

```

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

1.2. 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 1.2.7). Aby więc osiągnąć opóźnienie o długości 1 s przy zegarze HCLK o częstotliwości 72 MHz należy do funkcji `Delay` przekazać wartość 1535000.

1.2.8. 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 płycie 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 wyniknąć z programowo realizowanego opóźnienia. Podejście to zostanie poprawione w jednym z dalszych projektów.

1.2.9. 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 1.2 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 1.2), 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 (zakończanego 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. 1.2. Opis podłączenia oraz znaczenia poszczególnych pinów wyświetlacza LCD o 16 znakach i dwóch liniach

1.2.10. Konfiguracja PWM

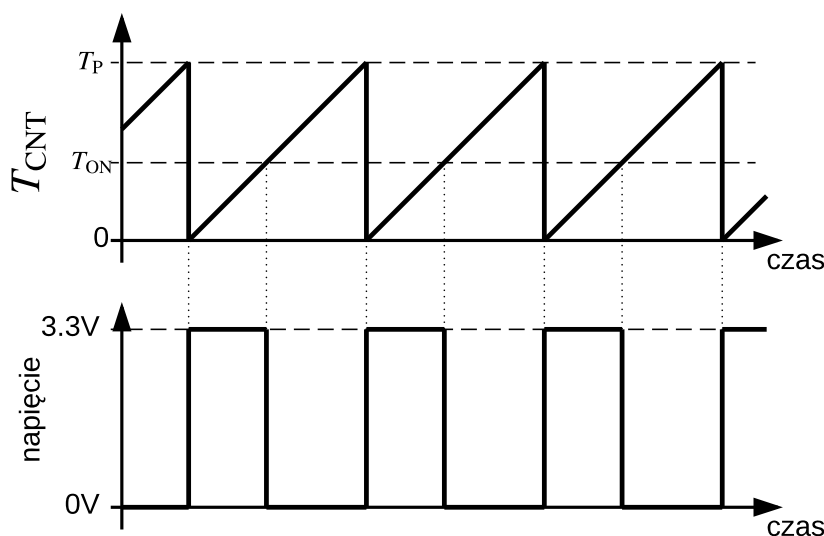
Ponieważ mikrokontrolery z rodziny STM32 są wysoce konfigurowalne, poniższy opis ograniczy się do omówienia wyłącznie konfiguracji układu timera w trybie generacji sygnału PWM. Aby wykorzystywać timery w tworzonym programie należy dodać je do konfiguracji poprzez odkomentowanie linijki

```
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. 1.27 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. 1.27. Uproszczony wykres zależności między zawartością licznika T_{CNT} , a postacią wygenerowanej fali PWM

1.2. 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. 1.28).

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 płytce 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. 1.28. 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 peryferali, 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 peryferali równoległe z pisanem do rejestrów mikrokontrolera jest możliwe i nierzadko stosowane. Dla tych, którzy wolą konsekwentnie trzymać się jednego rozwiązania: w standardowej bibliotece peryferali 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.

1.2.11. 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 0xFFF, natomiast wartość minimalna (0 V) jako 0x000. 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. 1.3 – 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. 1.29. 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.

1.3. 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. 1.29. Fragment noty katalogowej mikrokontrolera STM32F100, tabela 4. – rozpiska funkcji pinów

1.3. 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).

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

2.1. 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.

2.2. Treść ćwiczenia

2.2.1. Wykorzystane mechanizmy

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

```
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

2.2. TREŚĆ ĆWICZENIA

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 zwracany), 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 przerwającego 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 przerwania, które mogą nam przeszkodzić.

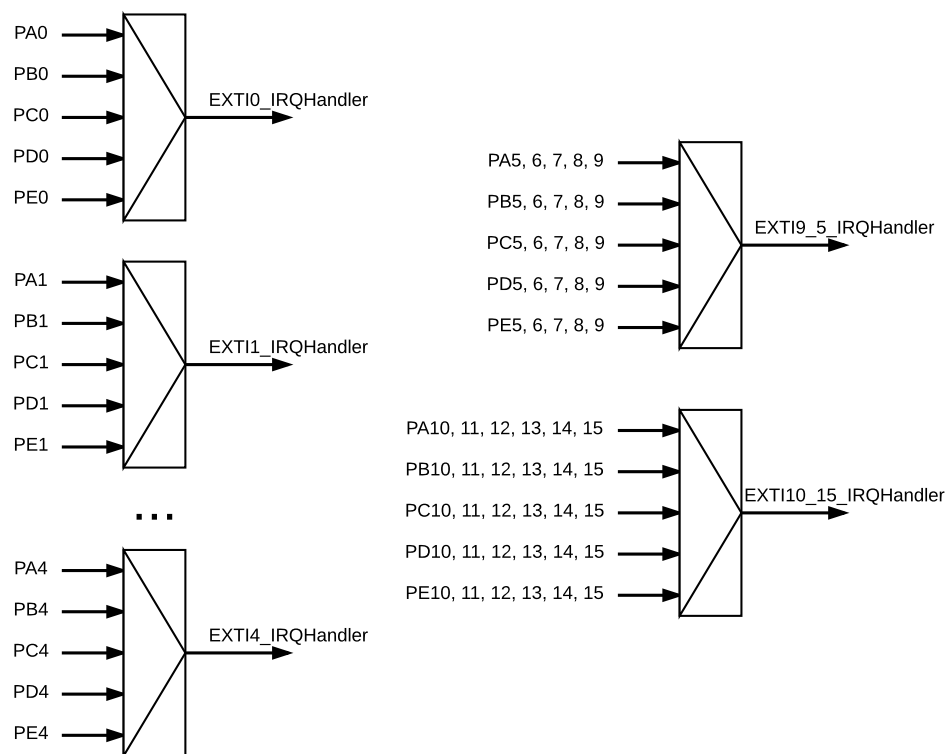
Poza tym, że przerwania powodują zatrzymanie wykonania głównej pętli programu, mogą także powodować zatrzymanie wykonania funkcji obsługującej inne przerwanie. Z tego powodu zostały wprowadzone priorytety przerw – im niższy priorytet przerwania tym jest ono ważniejsze. Warto tutaj zwrócić uwagę na wspomnianą 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.

Przerwania 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 wyłączania 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 przerwania 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 przerwaniach 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 przerwaniu 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.

Przerwania 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 bezpośrednio wywoływać pewną reakcję, np. wybudzić procesor, rozpocząć przetwarzanie sygnału analogowego na cyfrowy, itp. Co więcej niektóre układy

2.2. TREŚĆ ĆWICZENIA



Rys. 2.1. Schemat przypisywania linii do zgłaszanego przerwania

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 przerwania 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. 2.1. 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 EXTIO_5_IRQ (wspólne dla wszystkich portów). Analogicznie linie od 10 do 15 zgłaszają przerwanie EXTIO_15_IRQ.

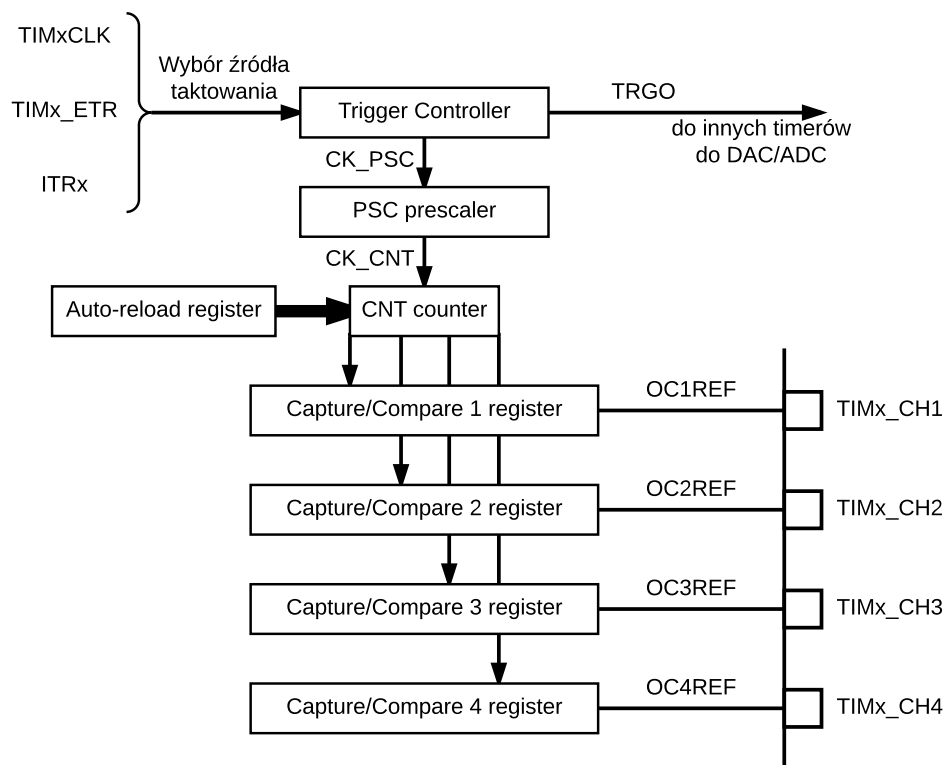
Timery: Timery, są to liczniki, których sygnałem wejściowym jest sygnał zegarowy. W mikrokontrolerach rodziny STM32 nie ma mowy o licznikach, lecz używa się właśnie pojęcia timer. W szczególnym przypadku timery służą do zliczania zboczy sygnału, który nie jest zegarowym (a więc działają jak zwykłe liczniki),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

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

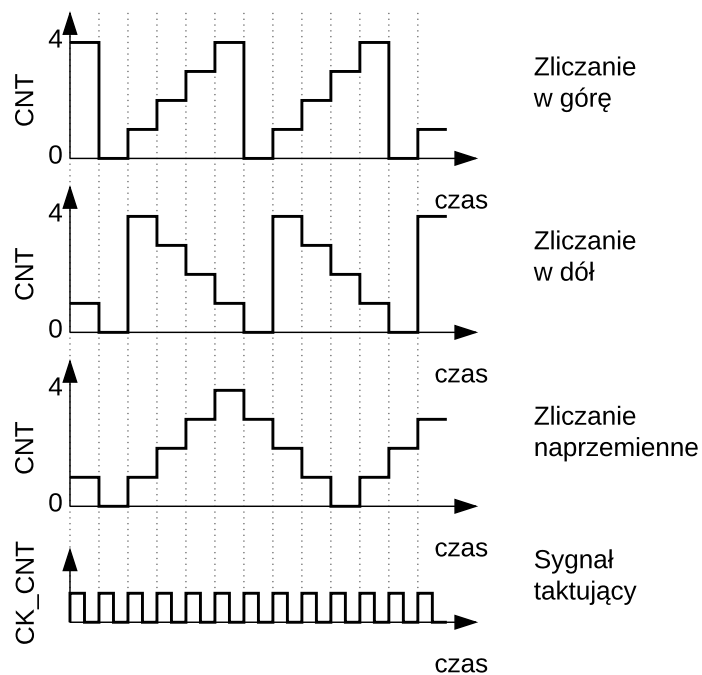
W tym ćwiczeniu w centrum uwagi będą dwa typy timerów: *Cortex System Timer* (nazywany często *SysTick*) oraz *General-purpose timer*. Pierwszy z nich należy do najprostszych w obsłudze (co wynika z niewielkich jego możliwości) timerów – zlicza on w dół takty sygnału zegarowego HCLK lub sygnału HCLK/8 (tj. sygnał zegarowy HCLK spowolniony ośmiokrotnie). Timer ten zlicza od zadanej 24-bitowej wartości do zera, po czym przeładowuje swój licznik ponownie zadaną 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 wyznaczone 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. 2.2. Na schemacie widać, że źródłem taktowania timera może być sygnał TIMxCLK (patrz Rys. 1.26), 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. 2.3. Przedstawiona została na nim zawartość licznika CNT w zależności od wybranego trybu zliczania, przy założeniu, że rejestr *Auto-reload* ma wartość 4. W trybie zliczania w 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

2.2. TREŚĆ ĆWICZENIA



Rys. 2.2. Schemat działania timera ogólnego przeznaczenia w trybie odmierzania czasu



Rys. 2.3. Zawartość licznika CNT w zależności od wybranego trybu zliczania

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

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

Prostą operację, jaką jest przełączenie bitu rejestru wyjściowego (TIMx.CHy), można wykonać więc na wiele sposobów: odpowiednią implementację obsługi przerwania (tryb *Timing*), wykorzystanie trybu *Toggle* lub jednego z trybów PWM. W poniższym ćwiczeniu będzie wykorzystany głównie tryb *Timing* – pozwoli to na późniejsze wykorzystanie wiedzy o implementacji obsługi 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.

2.2.2. 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:

```
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
```


2.2. TREŚĆ ĆWICZENIA

```
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
```

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 MHz, od około 0,54 Hz (okres około 1,86 s) do 9 MHz (okres około 111,11 ns). Tak więc aby zliczać pojedyncze sekundy należy koniecznie użyć sygnału wejściowego HCLK/8. W tym ćwiczeniu proponowane jest zliczanie kwantów 1 ms, a więc wybór zegara wejściowego nie jest krytyczny (na obu można zrealizować to

zadanie) – dla przykładu zostanie użyty sygnał HCLK/8. Tak więc **konfiguracja timera SysTick**, który zgłasza przerwanie co 1 ms wygląda następująco:

```
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 `msec` typu `static unsigned int`. Słowo kluczowe `static` w tym przypadku służy do ograniczenia widoczności zmiennej `msec` 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 większy od 0)
3 }
4
```

```
1 void Delay(unsigned int ms){
2     // zmiana wartosci licznika
3     // testowanie czy jest on większy 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.

Ustawienie podziału priorytetów przerwań: Przed rozpoczęciem pracy nad przerwami 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.

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 ); // włączenie przerw
10 TIM_Cmd(TIM4, ENABLE);                                   // aktywacja timera TIM4
11
12 // konfiguracja kanału 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 wywłaszczania
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 przerwanie, 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 konieczne 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.

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. 2.4 widoczny 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

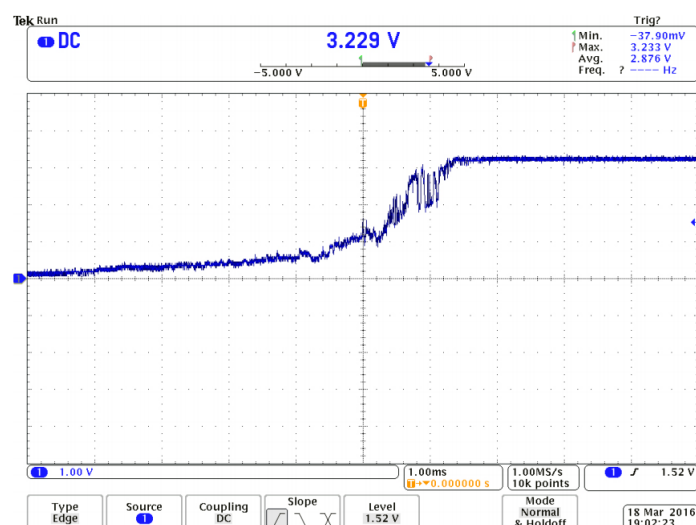
```

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 interakcji użytkownik-mikrokontroler, aby ten pierwszy miał świadomość, że mikrokontroler nie przestał działać, a jest po prostu bardzo zajęty realizacją niezmiernie ważnych zadań.

Wykorzystanie przerwań prowadzi jednak do innego problemu – reakcje na zmiany stanu przełącznika są niemal natychmiastowe. A więc widoczny na Rys. 2.4 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



Rys. 2.4. Oscylogram zjawiska drgań styków (wciśnięcie przełącznika)

sprzętowa realizacja filtru 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 peryferiali 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); // czyszczenie 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 przerwaniem przez niego generowanym):

```
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 przerw
7  TIM_Cmd(TIM3, DISABLE); // wyłączenie timera
8
9  NVIC_ClearPendingIRQ(TIM3_IRQn); // czyszczenie bitu przerw.
10 NVIC_EnableIRQ(TIM3_IRQn); // włączenie przerwania
11 NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn; // nazwa przerwania
12 NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1; // prior. wywłaszczania
13 NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1; // podpriorytet
14 NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE; // aktywowanie konfigur.
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**:

2.2. TREŚĆ ĆWICZENIA

```
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:

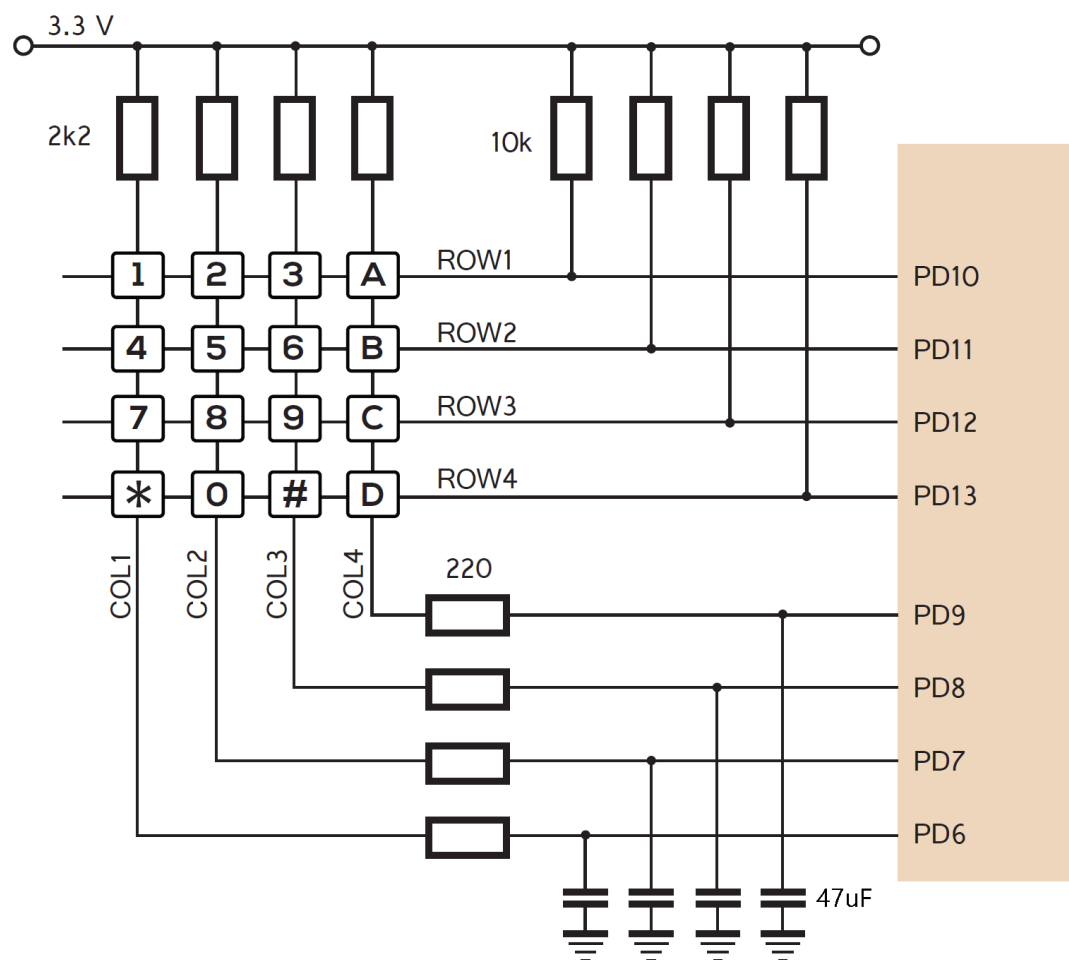
```
1 void TIM3_IRQHandler(void){
2     if(TIM_GetITStatus(TIM3,TIM_IT_Update) != RESET){ // sprawdzenie przyczyny
3         TIM_ClearITPendingBit(TIM3, TIM_IT_Update);    // wyczyszczenie bitu przerw.
4         if(GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0) == Bit_RESET) // jeśli wciśnięty
5             LED(LED5,LED_TOGGLE);                     // zrob cos (przelacz 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.

Obsługa klawiatury numerycznej Do zestawu ZL27ARM można bardzo łatwo podpiąć klawiaturę o 4 kolumnach i 4 wierszach (Rys. 2.5). 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



Rys. 2.5. Klawiatura 4×4



Rys. 2.6. Schemat podłączenia klawiatury 4×4 (źródło: *Systemy Mikroprocesorowe w Sterowaniu. Część I: ARM Cortex-M3*)

kondensatorów), które zostały wprowadzone do omawianej klawiatury w postaci osobnej płytki łączącej zestaw uruchomieniowy i klawiaturę. Schemat usprawnionej klawiatury jest przedstawiony na Rys. 2.6. 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 **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.

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

2.2. TREŚĆ ĆWICZENIA

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 ustawien 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
9     ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; // pomiar wyrównany do prawej
10    ADC_InitStructure.ADC_NbrOfChannel = 1; // jeden kanał
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 wybór przerwania TIM_IT_CC2 jako sygnału rozpoczynającego przetwarzanie, zmiana kanału, który będzie wykorzystany do pomiarów i co za tym idzie dodatkowa linijka uruchamiająca czujnik temperatury znajdujący się w samym mikrokontrolerze. Oczywiście należy również pamiętać o aktywowaniu i konfiguracji przerwania wyzwalanego na zakończenie konwersji ADC_IT_EOC. W obsłudze tego przerwania należy, tak jak zawsze, sprawdzić, co 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 mikro-

kontrolerów w wymagającym środowisku, np. małej zamkniętej obudowie telefonu, gdzie głównym źródłem ciepła jest właśnie sam mikrokontroler.

Dodatkowe informacje: Warto pamiętać o kolejności konfiguracji kolejnych peryferiali:

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

2.3. 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.

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

3.1. 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.

3.2. Treść ćwiczenia

3.2.1. 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.

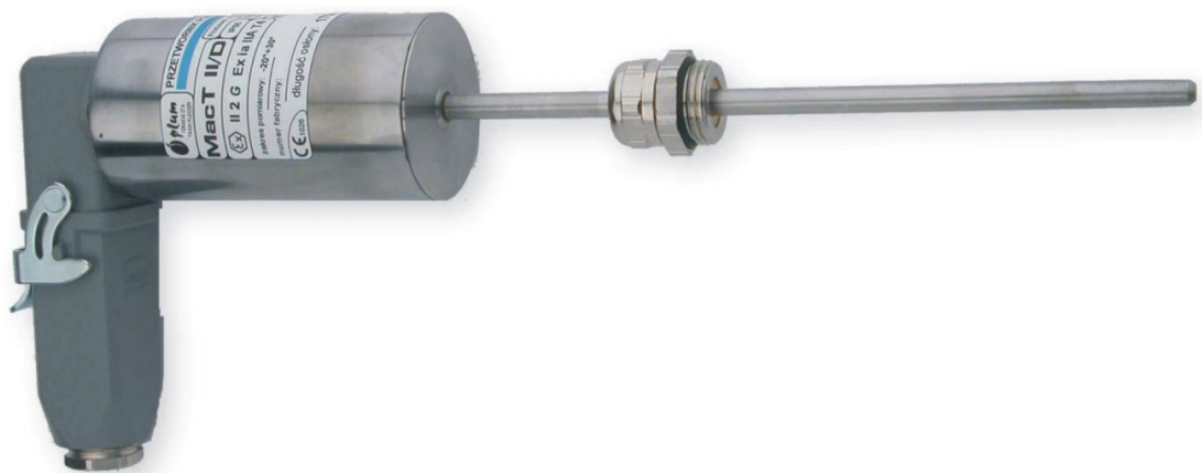
W związku z powyższym przemysłowe czujniki mogą być podłączone bardzo długimi kablami do urządzeń odczytujących pomiary. Pomiar z wykorzystaniem napięcia w takiej sytuacji byłby obciążony znaczącym błędem.

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

3.2.2. 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. 3.1). 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. 3.2. Należy zwrócić szczególną uwagę na podłączenie do mikrokontrolera – zamiana masy płytki z pinem wejściowym może spowodować uszkodzenie płytki rozwojowej. Dla ułatwienia podłączeń



Rys. 3.1. Cyfrowy przetwornik temperatury typu MacT II firmy Plum.

3.2. TREŚĆ ĆWICZENIA

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:

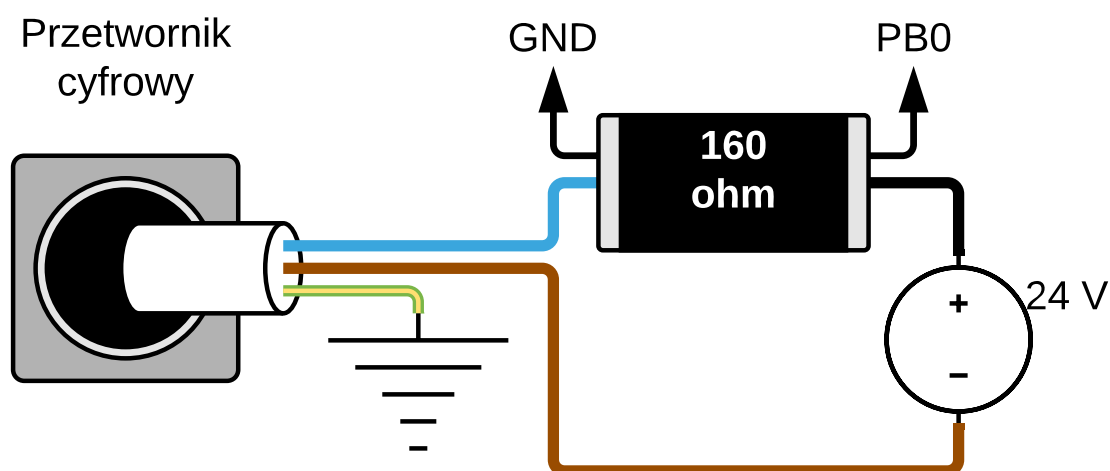
$$U(t) = \left(\frac{t + 30}{90}16\text{ mA} + 4\text{ mA} \right) 160\Omega = \frac{t + 30}{90}2,56\text{ V} + 0,64\text{ V}$$

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. 3.2. 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.

3.2.3. 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*.

3.2. TREŚĆ ĆWICZENIA

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. 3.3 (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.

3.2.4. 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. 3.3. 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);
```

3.2.5. 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 sugestie – rozdział 2.4.1 dokumentu *MODBUS over serial line...*)

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

3.2. 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);
```


3.2. TREŚĆ ĆWICZENIA

```
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 nasłuchiwanie 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 $50\mu\text{s}$. 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.

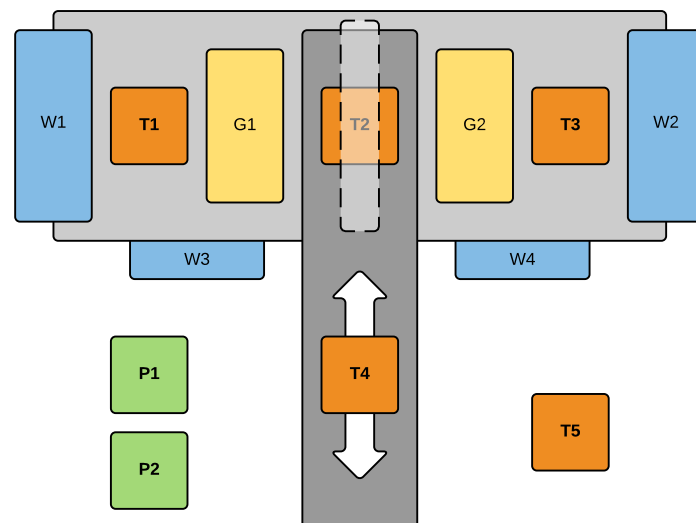
Procedura inicjalizacji komunikacji z użyciem protokołu `MODBUS` składa się z następujących etapów:

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.

3.2.6. 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. 3.4. 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. 3.4. 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.

3.3. 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 4-20 mA 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.

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