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

## 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 jednoukładowych. 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. 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 zostanie przez prowadzącego laboratorium.

### 1.2.1. STM32CubeIDE

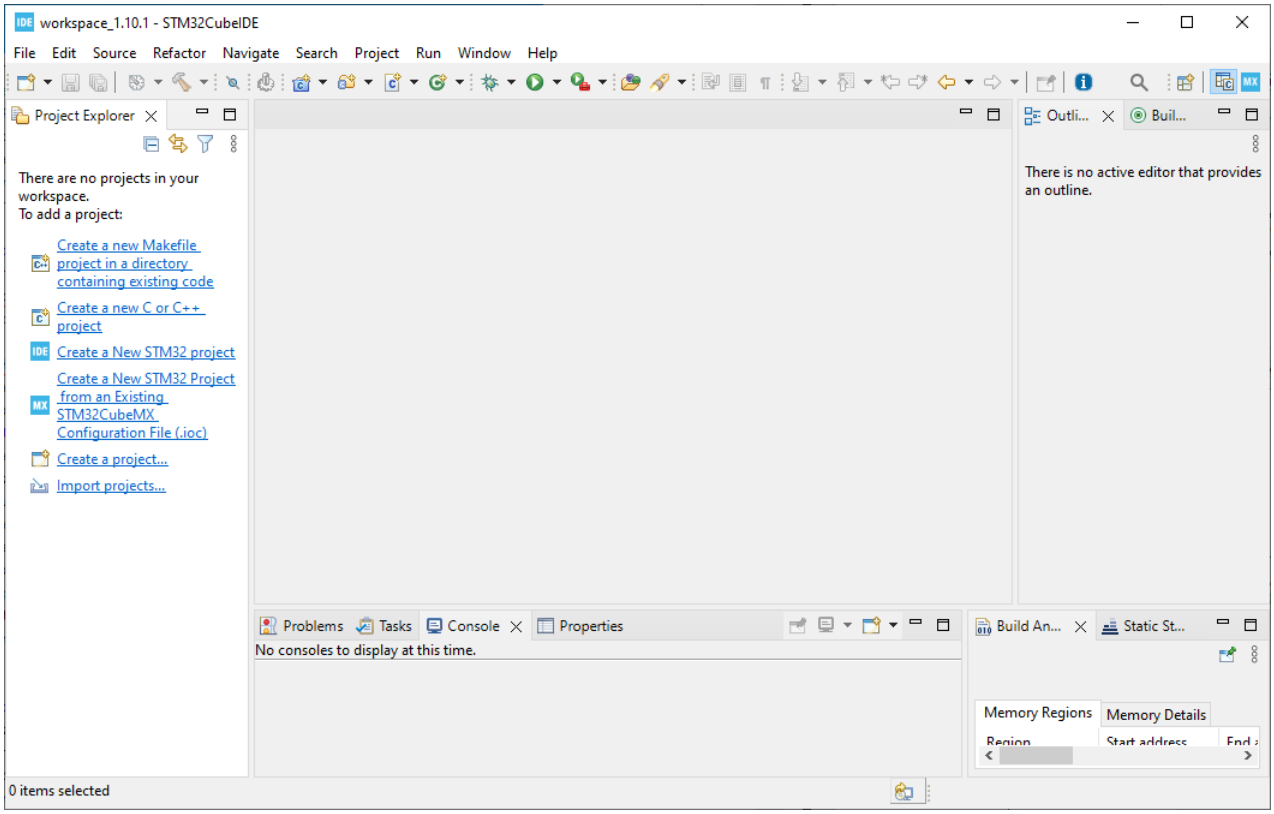
Zintegrowane środowisko deweloperskie o nazwie STM32CubeIDE bazuje na oprogramowaniu Eclipse IDE. Powoduje to, że często w trakcie korzystania z tego oprogramowania może pojawiać się odniesienie do nomenklatury związanej z tym środowiskiem, takie jak „perspektywa" czy „katalog roboczy". Perspektywa,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

### 1.2.2. Podłączenie mikrokontrolera

Jako mikrokontroler użyty zostanie STM32F103VBT6, zawierający się w zestawie uruchomieniowym ZL27ARM. Ogólne informacje dotyczące tego zestawu znajdują się w Instrukcji Użytkownika związanej z tym zestawem, 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 (rys. 1.2) odbywać się będzie za pośrednictwem programatora *J-LINK* (firmy *SEGGER*) w wersji edukacyjnej (*EDU*) – rys. 1.3. Jest on podłączany poprzez złącze *JTAG* (*Joint Test Action Group*), które pozwala na testowanie (w tym debugowanie i śledzenie wykonania programu) procesora wlutowanego w zmontowaną płytę drukowaną. Połączenie między zestawem ZL27ARM a programatorem *J-LINK EDU* następuje przy użyciu 20-żyłowego kabla, który z jednej strony jest




Rys. 1.1. Okno środowiska STM32CubeIDE


Rys. 1.2. Płytka rozwojowa ZL27ARM z mikrokontrolerem STM32F103VBT6

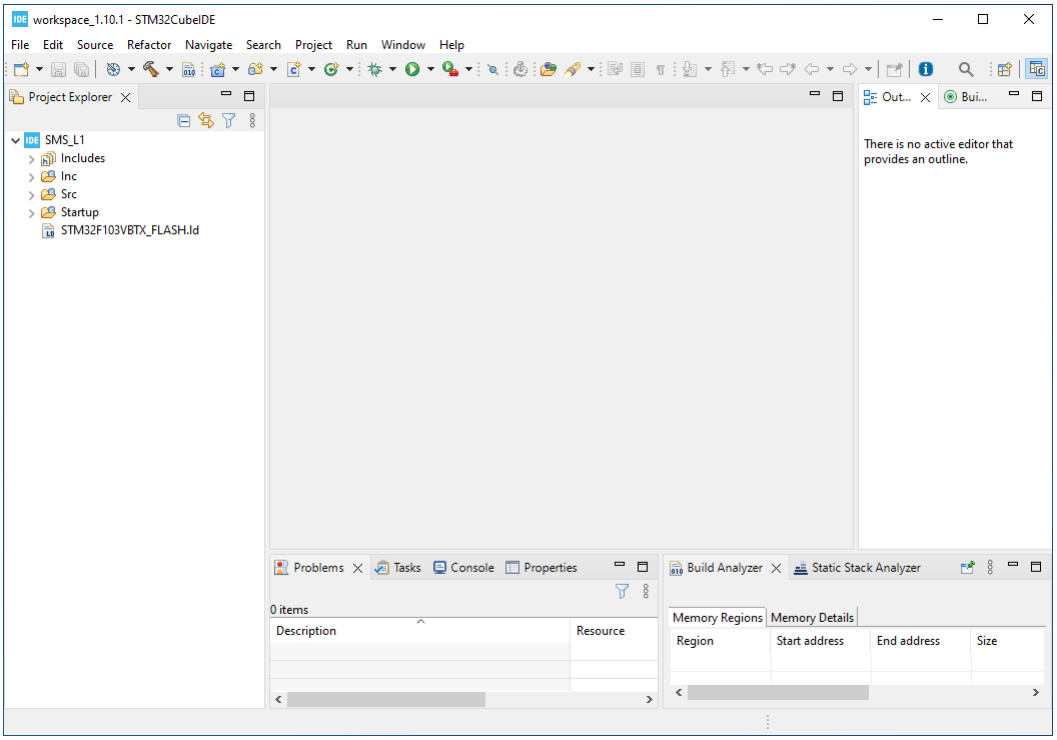

Rys. 1.3. Programator firmy SEGGER, *J-LINK EDU*

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.

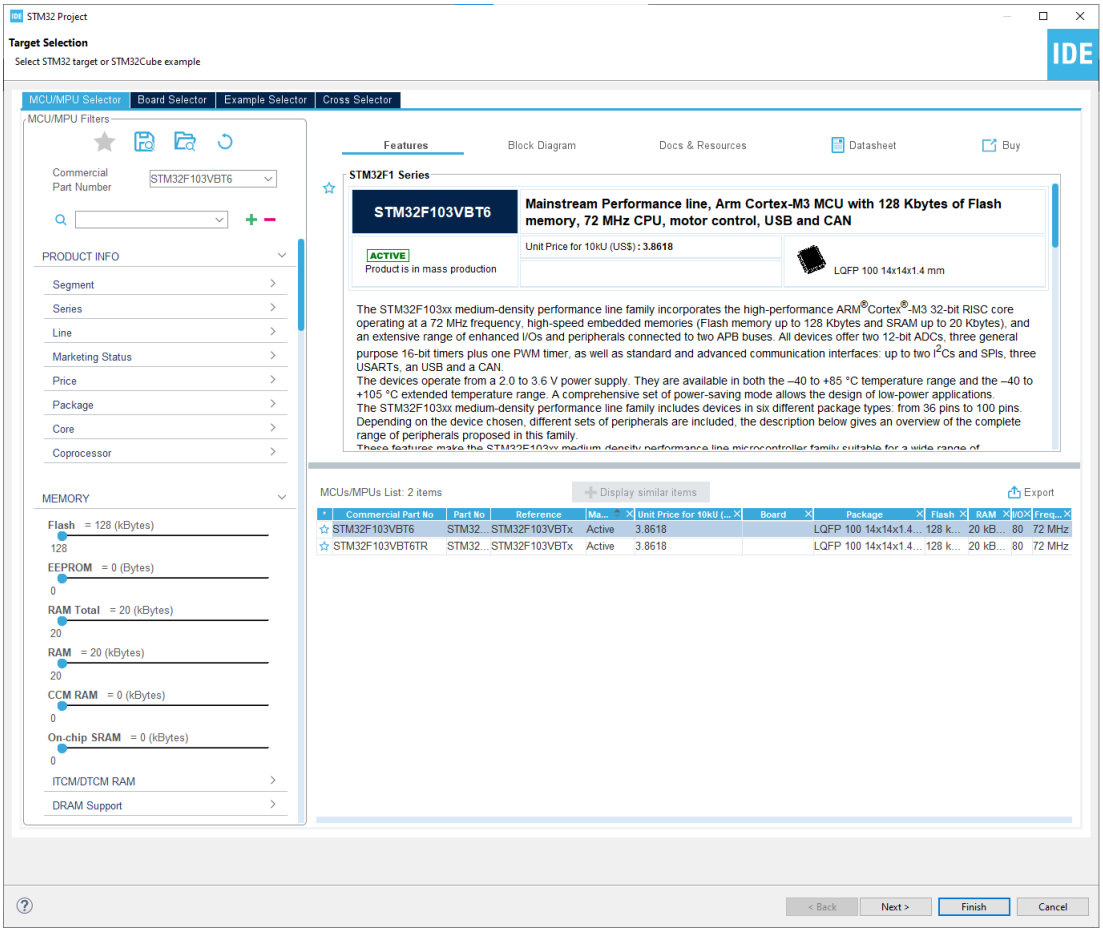
### 1.2.3. Stworzenie projektu

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. 1.5). Należy wybrać odpowiedni i wcisnąć przycisk *Next*. W kolejnym oknie (rys. 1.6) 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. 1.4.

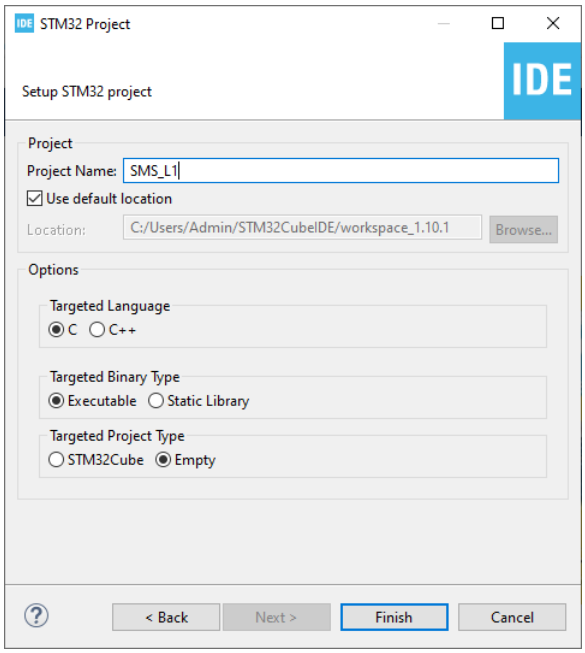
Obecnie przygotowany kod pozwala na tworzenie projektów z wykorzystaniem pisania po rejestrach. Jest to metoda pisania oprogramowania na mikrokontrolery rekomendowana przez wielu programistów, jako, że zapewnia niemal absolutną kontrolę nad wykonaniem programu. W ramach tych ćwiczeń nie będziemy skupiali się na tym podejściu, ponieważ




Rys. 1.4. Okno środowiska STM32CubeIDE bezpośrednio po stworzeniu pierwszego projektu



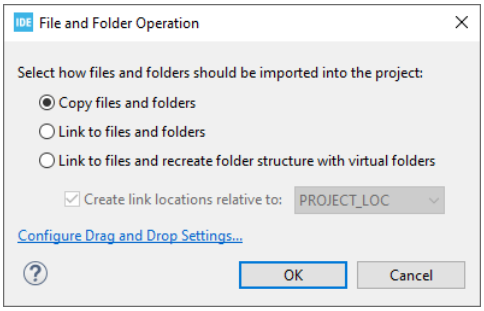

Rys. 1.5. Okno wyboru platformy docelowej (Target Selection)



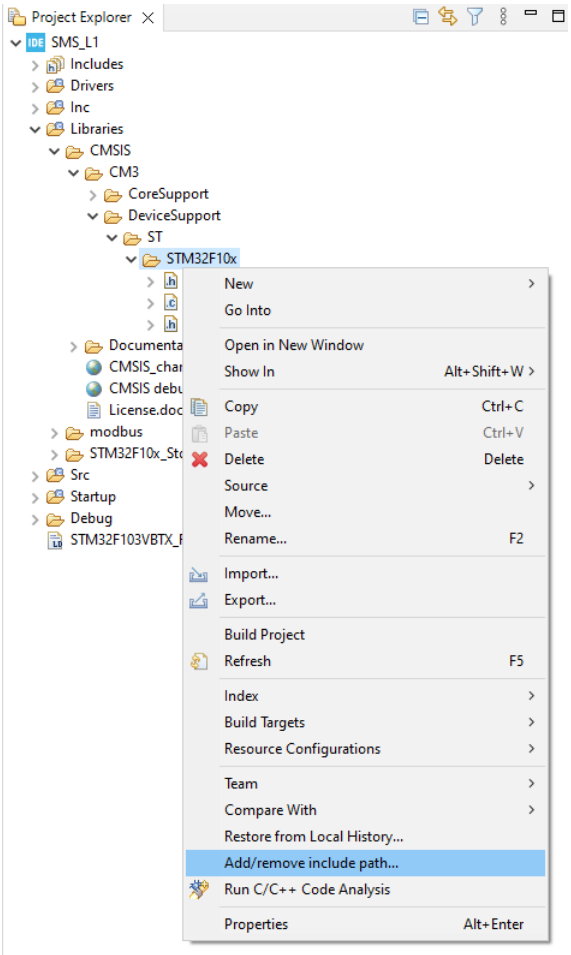

Rys. 1.6. Okno podstawowej konfiguracji projektu

zakłada ono doskonałą znajomość dokumentacji rozważanego mikrokontrolera, natomiast gotowy kod jest nieprzenoszalny.

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 ukaże się okno (rys. 1.7) z pytaniem czy skopiować pliki czy wyłącznie je dowiązać, w któ-




Rys. 1.7. Okno wyboru sposobu importowania katalogów z plikami do projektu



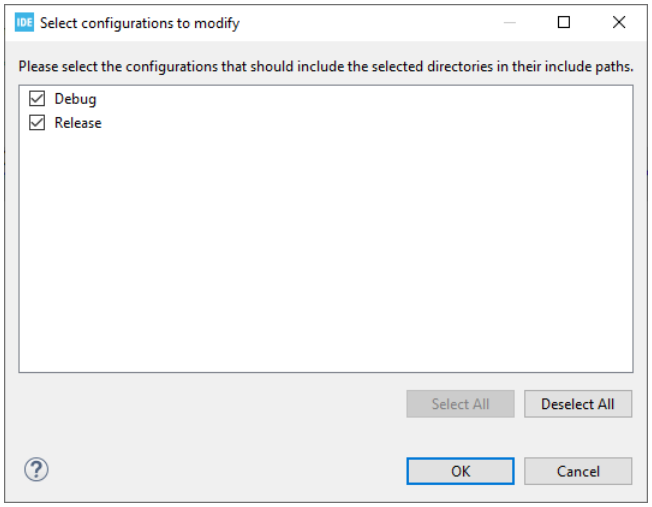

Rys. 1.8. Menu kontekstowe pozwalające na dodanie katalogu projektu do listy ścieżek z plikami nagłówkowymi

rym 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. 1.8), a w oknie, które się pojawiło (rys. 1.9) 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:

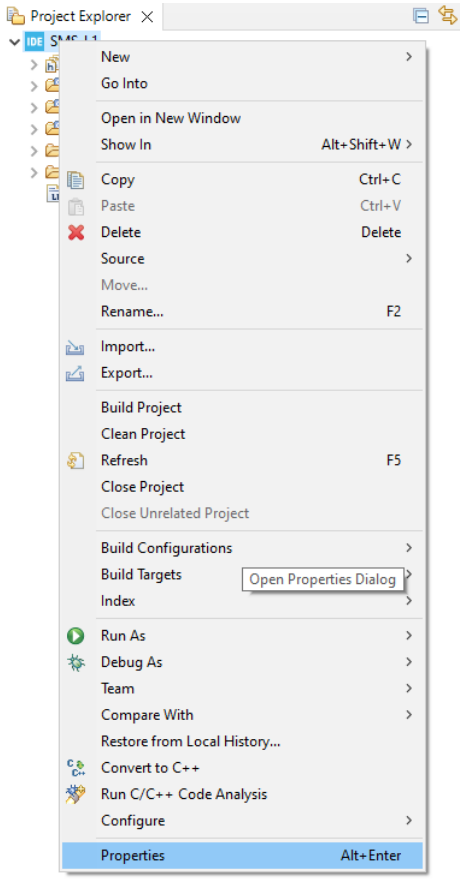
— Libraries/CMSIS/CM3/CoreSupport
— Libraries/CMSIS/CM3/DeviceSupport/ST/STM32F10x
— Libraries/STM32F10x_StdPeriph_Driver/inc

Następnie należy wskazać kompilatorowi, że w dodanych katalogach znajdują się pliki, które powinny zostać skompilowane. W tym celu należy kliknąć prawym przyciskiem myszy na korzeń projektu (rys. 1.10) 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 zakładaka *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. 1.11). Tutaj należy kliknąć przycisk *Add Folder...* a następnie wybrać katalogi *Libraries* oraz *Drivers* (rys. 1.12) i wcisnąć przycisk OK. Kompilator przeszuka nie tylko te konkretne katalogi, ale także wszystkie zagnieżdżone katalogi w tych katalogach.

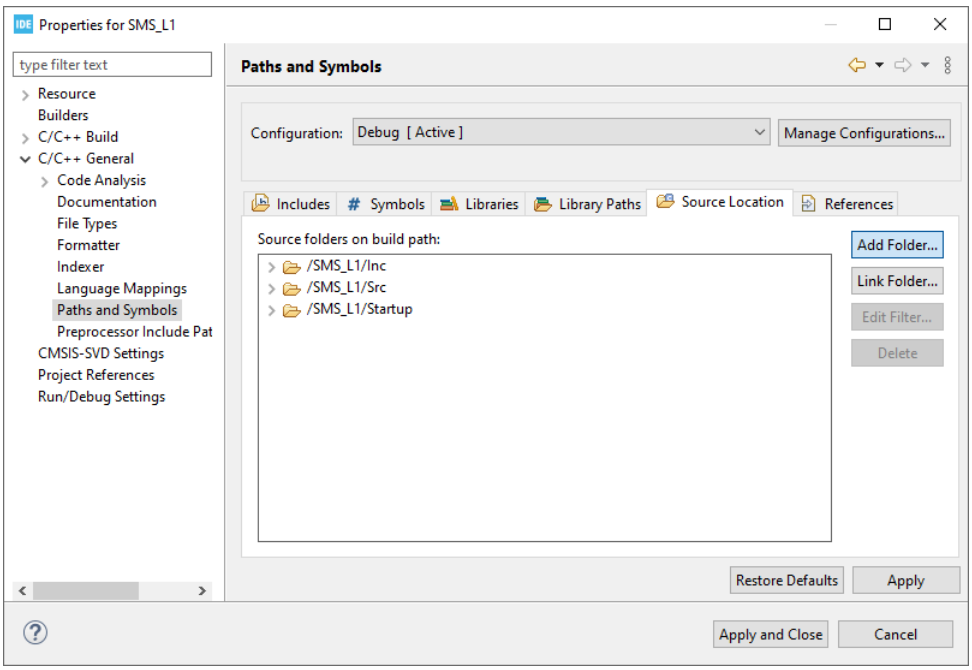
Należy także dodać dwa symbole preprocesora wykorzystywane w plikach nagłówkowych biblioteki SPL. W tym celu należy przejść do zakładki *Symbols* (rys. 1.13) i po kliknięciu przycisku *Add...* wpisać w pole *Name:* wyrażenie `USE_STDPERIPH_DRIVER`, zaznaczyć *Add to all configurations* (rys. 1.14), po 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*



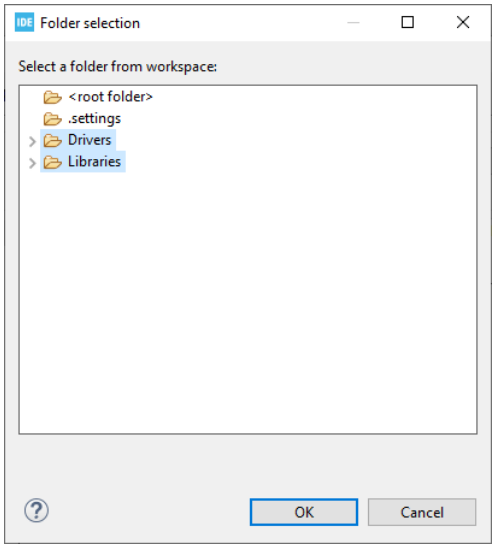

Rys. 1.9. Okno wyboru konfiguracji dla której pliki nagłówkowe mają być dodane



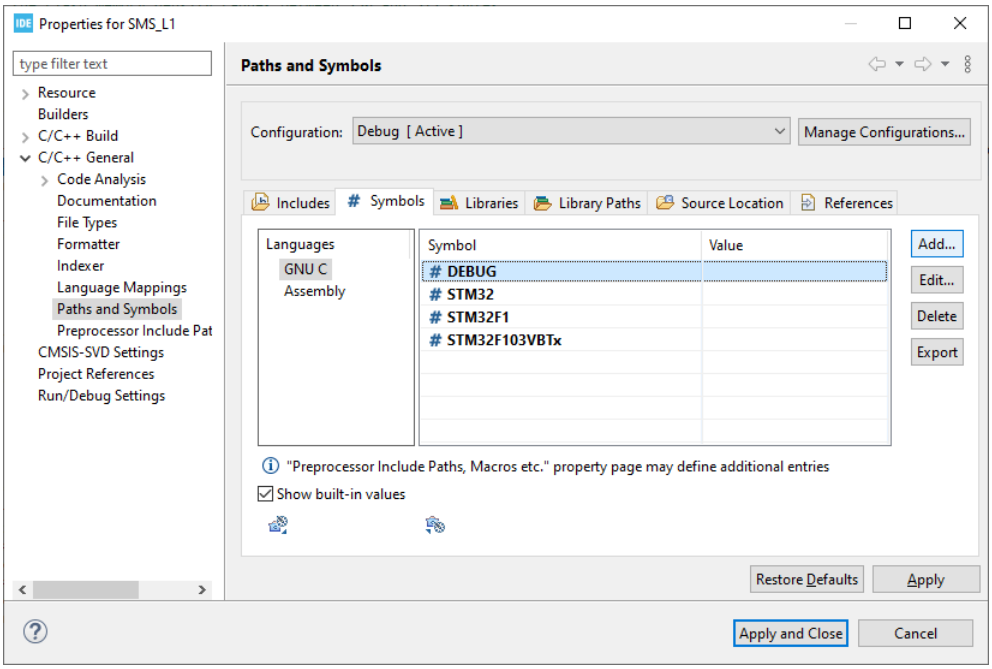

Rys. 1.10. Menu kontekstowe projektu pozwalające na otwarcie szczegółowych ustawień projektu



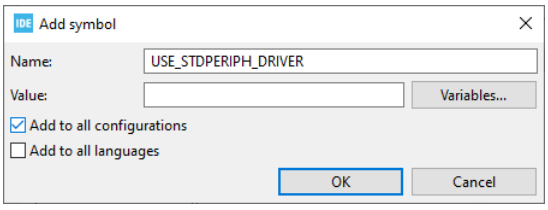

Rys. 1.11. Okno szczegółowych ustawień projektu – ścieżki z plikami źródłowymi




Rys. 1.12. Okno pozwalające na wybór katalogów z plikami źródłowymi




Rys. 1.13. Okno szczegółowych ustawień projektu – symbole preprocesora

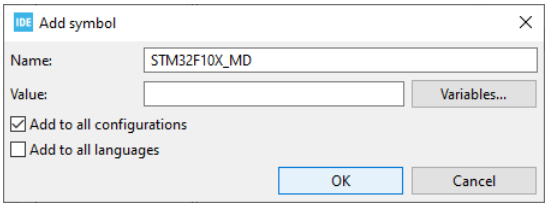



Rys. 1.14. Okno dodawania nowego symbolu preprocesora – symbol informujący kompilator o wykorzystaniu bibliotek SPL

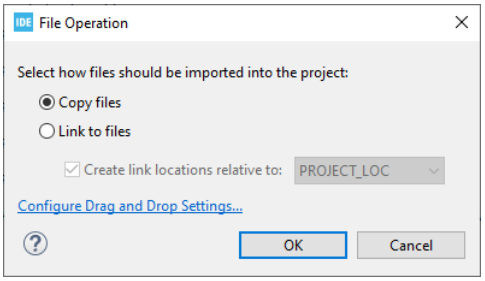
*all configurations* (rys. 1.15), po czym wcisnąć *OK*. Następnie można zamknąć to okno przyciskiem *Apply and Close*.

Przy próbie kompilacji na tym etapie, kompilator zwróci uwagę na brak pliku o nazwie `stm32f10x_conf.h`, który to jest plikiem konfigurującym 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. 1.16, gdzie należy wybrać kopiowanie (pierwszą opcję) i zaakceptować to przyciskiem *OK*. W tym momencie program powinien się skompilować z powodzeniem, co w widoku konsoli powinno wyglądać podobnie jak na rys. 1.17.

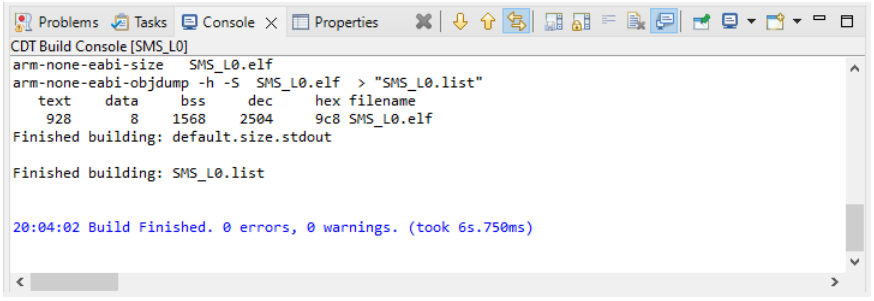
Na potrzeby dalszych rozdziałów warto jeszcze dodać możliwość wykorzystania liczb zmiennoprzecinkowych w funkcji `sprintf`. W tym celu należy ponownie wejść w opcje projektu (rys. 1.10), w drzewie z lewej strony wybrać *C/C++ Build→Settings*, następnie w zakładce *Tool Settings* wybrać w drzewie *MCU Settings*. Tutaj należy wybrać *Use float with printf from newlib-nano (-u _printf_float)* i wcisnąć przycisk *Apply and Close* (rys. 1.18).



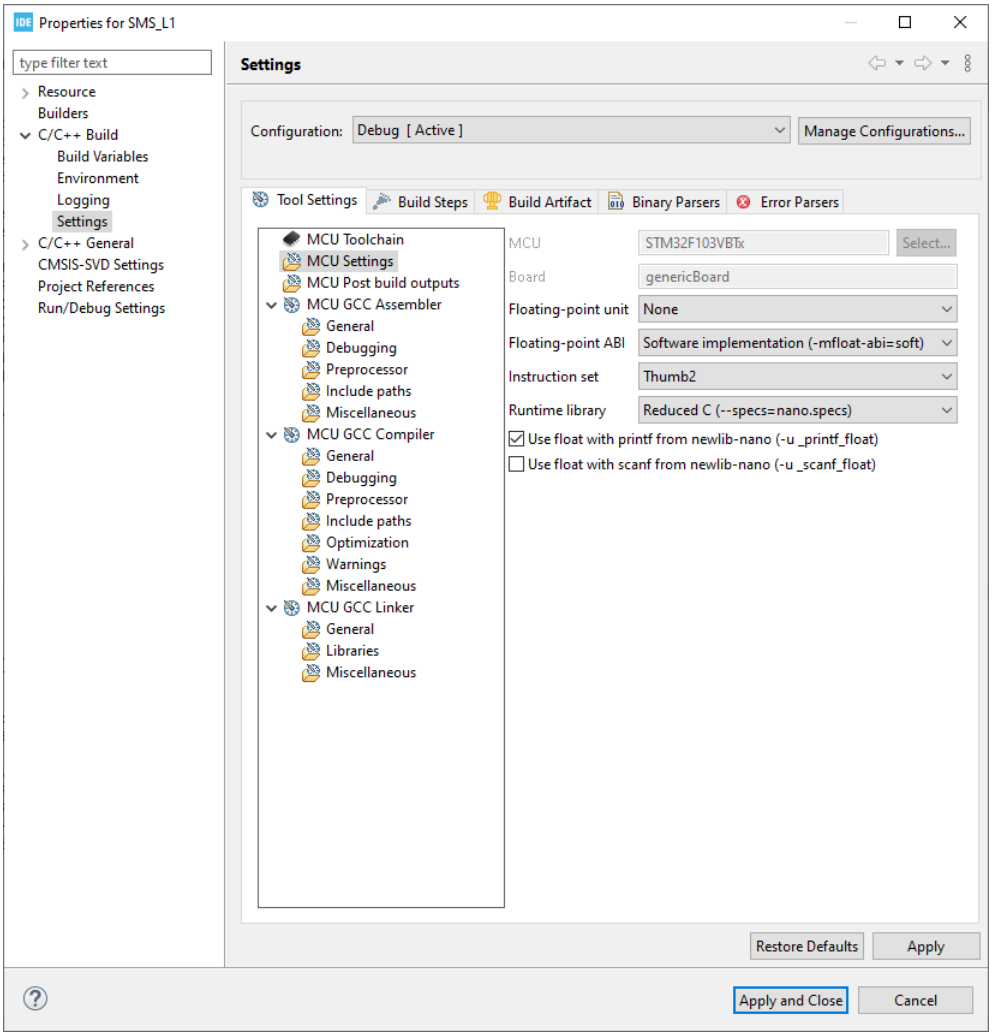

Rys. 1.15. Okno dodawania nowego symbolu preprocesora – symbol informujący kompilator o wykorzystaniu instrukcji dla mikrokontrolera o „średniej gęstości” (*Medium Density*)



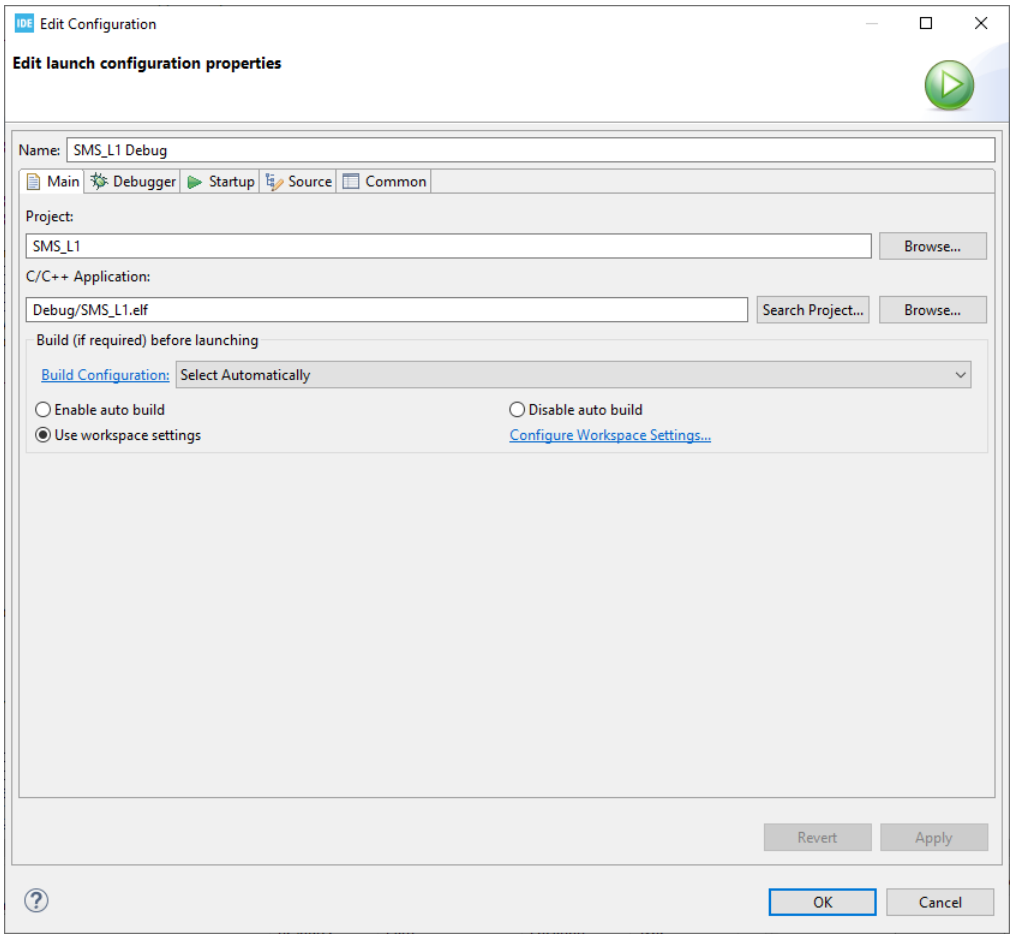

Rys. 1.16. Okno wyboru sposobu importowania plików do projektu



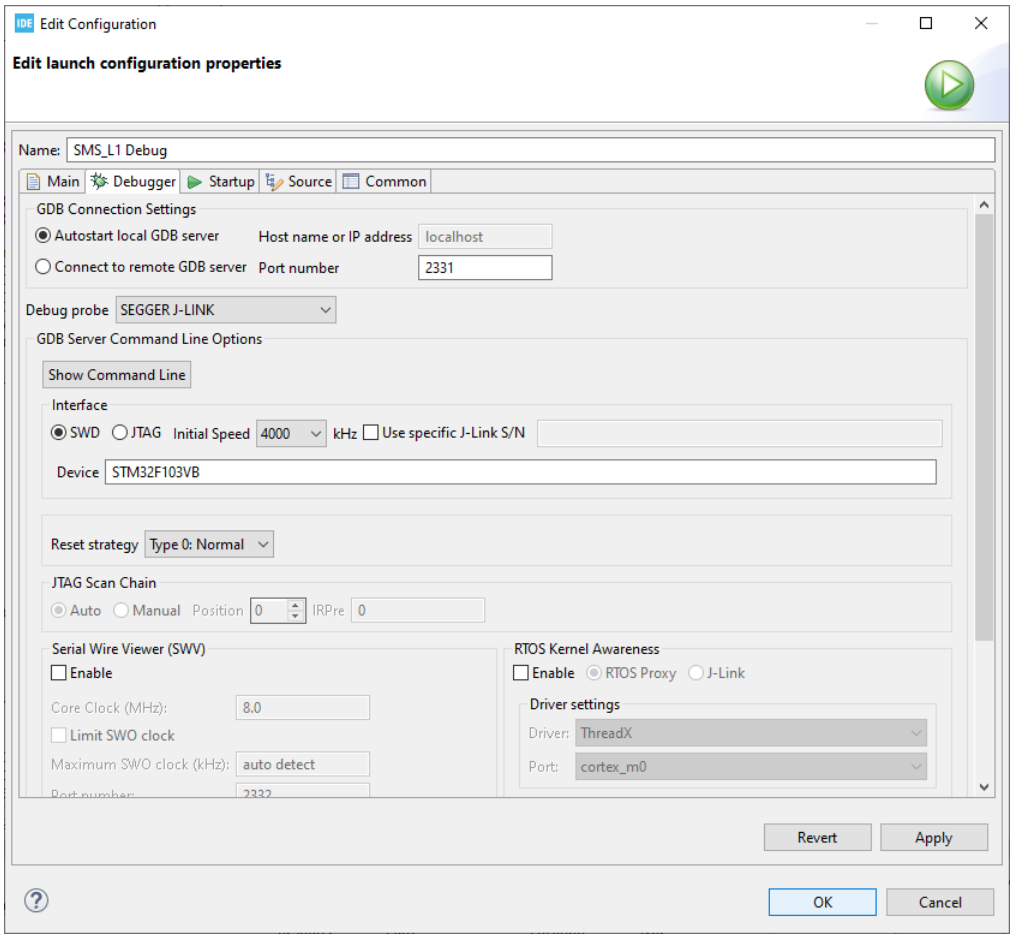

Rys. 1.17. Widok konsoli informujący o poprawnym przebiegu kompilacji



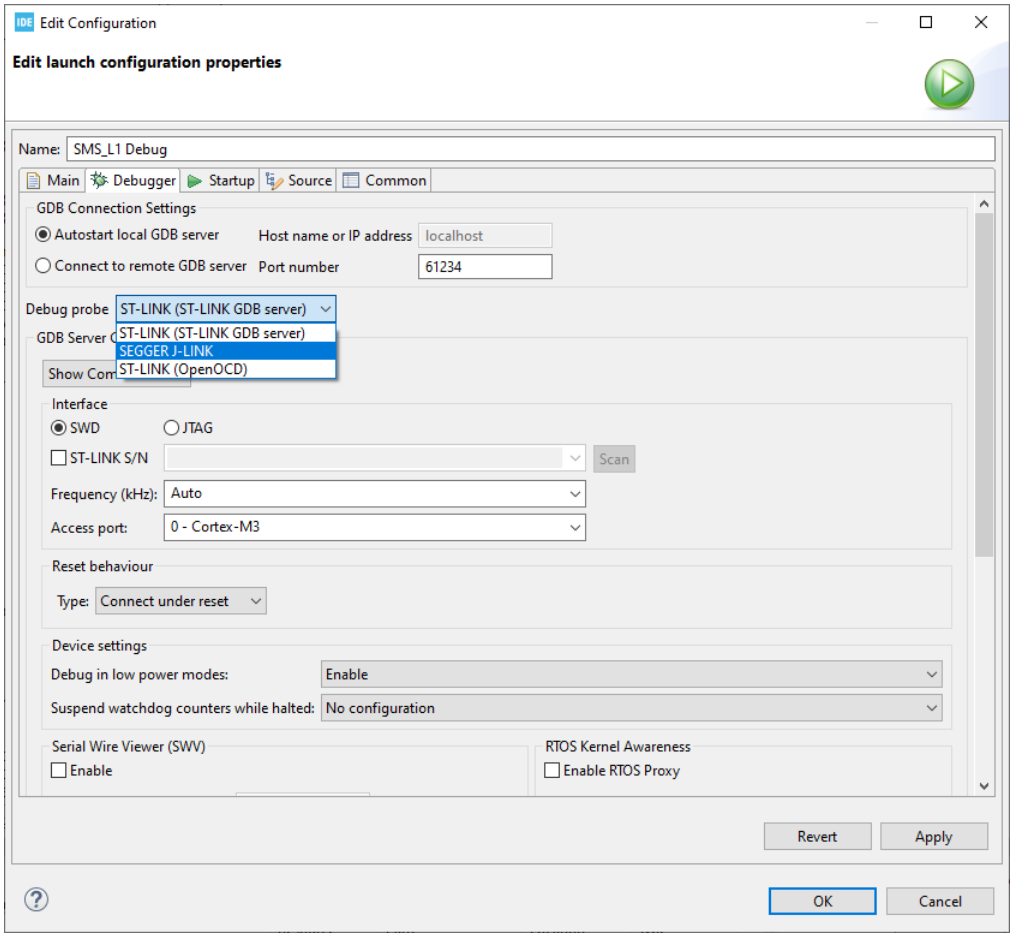

Rys. 1.18. Okno szczegółowych ustawień projektu – przydatne ustawienia kompilacji



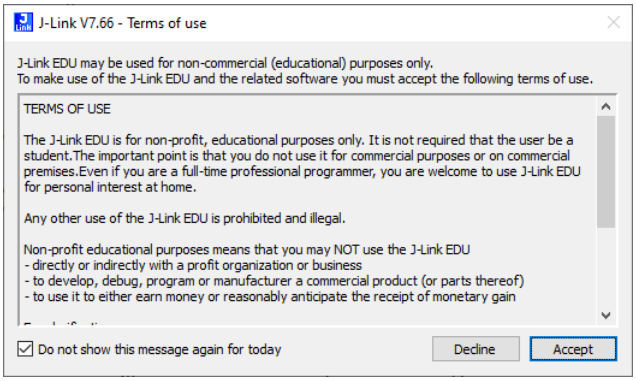

Rys. 1.19. Okno konfiguracji sposobu wgrywania programu na mikrokontroler – zakładka główna




Rys. 1.20. Okno konfiguracji sposobu wgrywania programu na mikrokontroler – zakładka konfiguracji debuggera z domyślną konfiguracją debuggera SEGGERJ-LINK




Rys. 1.21. Okno konfiguracji sposobu wgrywania programu na mikrokontroler – zakładka konfiguracji debuggera




Rys. 1.22. Okno informujące o warunkach licencji na programator w wersji edukacyjnej

### 1.2.4. Wgranie programu na mikrokontroler

Taki program można już wgrać na mikrokontroler, co wykonuje się poprzez kliknięcie przycisku zielonego kółka z białą strzałką na pasku narzedzi lub poprzez menu *Run→Run As→1 STM32 C/C++ Application*, co spowoduje pojawienie się okna widocznego na rys. 1.19. W tym oknie zdefiniowana jest nazwa konfiguracji wykorzystanej do wgrywania oprogramowania na mikrokontroler. Interesującą nas zakładką jednak będzie zakładka *Debugger* (rys. 1.21), gdzie należy zmienić programator na odpowiedni. W tym celu należy w polu *Debug probe* 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. 1.3. Po wybraniu odpowiedniego programatora powinno ukazać się okno z dodatkowymi opcjami jak na rys. 1.20 – 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. 1.22). 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.

### 1.2.5. Konfiguracja sprzętowa mikrokontrolera

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

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

Przekłada się to na następujące ustawienia zworek na płycie ZL27ARM:

| Nazwa | Pozycja |
|---|---|
| *PWR_SEL* | USB |
| *BOOT0* | 0 |
| *BOOT1* | 0 |
| *LEDs* | OFF |
| *LCD_PWM* | OFF |
| *USB* | OFF |

Ponieważ mikrokontroler nie jest zasilany przez programator, należy go podłączyć kablem USB do źródła zasilania (np. komputera). W tym celu (przy przełączniku zasila-

nia *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*.

### 1.2.6. Przebieg laboratorium (samodzielnie wykonywane zadanie)

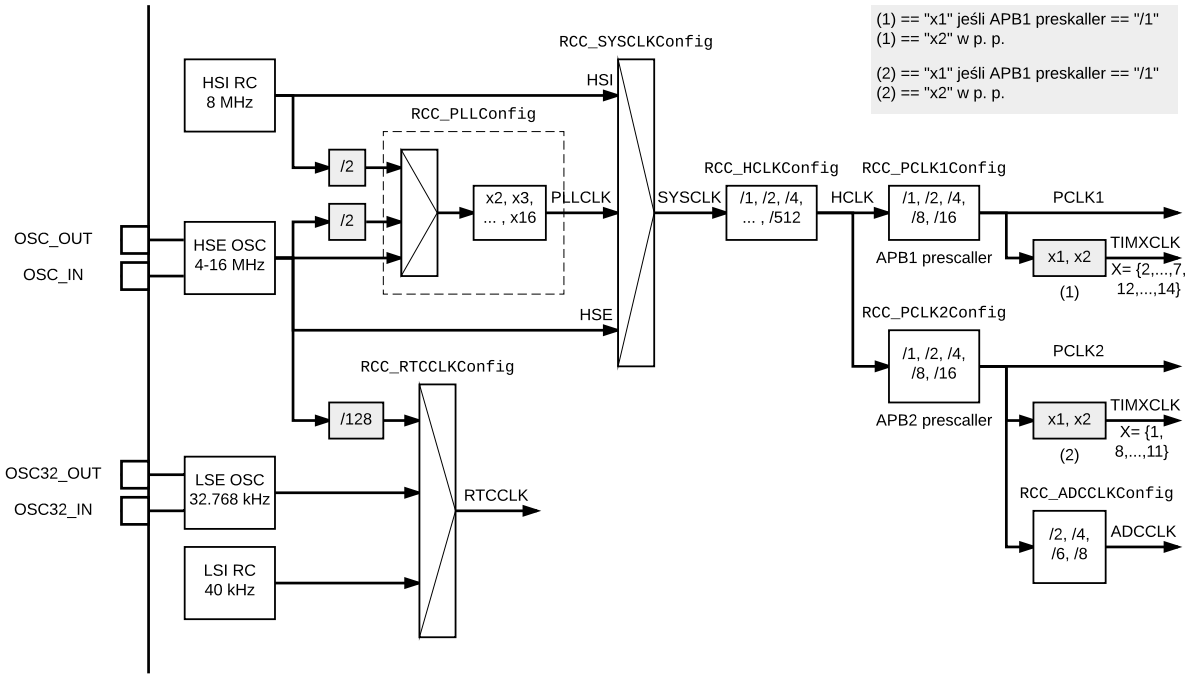
Mikrokontroler składa się z wielu (tysięcy) bramek logicznych, które pracują w trybie synchronicznym. Oznacza to, że są one taktowane zegarem i wraz z nim zmienia się wartość na wyjściu bramek. Dzięki temu unika się takich problemów jak zjawisko hazardu lub wyścigu, a więc problemów wynikających z niezerowego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

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

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

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




Rys. 1.23. Uproszczony schemat układu taktowania procesora i peryferiali – pełna wersja znajduje się w dokumencie RM0008 (Rys. 8)

cjach z powodzeniem wykorzystywane, warto rozważyć użycie tanich, znacznie dokładniejszych rezonatorów kwarcowych. Ponadto moduł RCC jest wyposażony w konfigurowalne dzielniki częstotliwości i pętlę *Phase Locked Loop* (PLL) – można ją utożsamiać z „mnożnikiem częstotliwości". Uproszczony schemat układu taktowania procesora i peryferiali widoczny jest na rys. 1.23. Na schemacie są dodatkowo umieszczone nazwy funkcji ze standardowej biblioteki do obsługi peryferiali, które pozwalają na konfigurację poszczególnych elementów i sygnałów taktujących (nazwy te są zapisane czcionką stałoszerokościową).

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:

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

Tak skompilowany projekt jest punktem wyjścia do konfiguracji zegarów (w oparciu o drzewo zegarów z rys. 1.23), 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 $0 < \text{SYSCLK} \leqslant 24\,\text{MHz}$
— `FLASH_Latency_1` jeśli $24 < \text{SYSCLK} \leqslant 48\,\text{MHz}$
— `FLASH_Latency_2` jeśli $48 < \text{SYSCLK} \leqslant 72\,\text{MHz}$

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

```
/****************************************
 * projekt01: konfiguracja zegarow      *
 ****************************************/
#include "stm32f10x.h"

#define DELAY_TIME 1535000

void RCC_Config(void);
void GPIO_Config(void);
void LEDOn(void);
void LEDOff(void);
void Delay(unsigned int);

int main(void) {
  RCC_Config();       // konfiguracja RCC
  GPIO_Config();        // konfiguracja GPIO

  while(1) {            // petla glowna programu
    LEDOn();            // wlaczenie diody
    Delay(DELAY_TIME); // odczekanie 1s
    LEDOff();           // wylaczenie diody
    Delay(DELAY_TIME); // odczekanie 1s
  }
}

```

W powyższym kodzie należy zmodyfikować wartość stałej `DELAY_TIME` zgodnie z tabelą 1.2.6, 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 zworkę na pozycję `On` i ponownie włączyć mikrokontroler. Diody najprawdopodobniej rozświetlą się z czasem mimo braku jakiejkolwiek interakcji ze strony użytkownika. Jest to ciekawe zjawisko wynikające z niepodciągnięcia wyjść prowadzących do diod, które niestety nie zostanie tutaj szczegółowo omówione. Należy jednak pamiętać, że zjawisko to ma wpływ wyłącznie na piny, które nie są skonfigurowane jako wyjścia – na tę chwilę nie należy się tym przejmować.

Poniżej została przedstawiona przykładowa funkcja konfigurująca zegary na ich maksymalne dozwolone wartości (dla mikrokontrolera STM32F103VB są to: HCLK = 72 MHz, PCLK1 = 36 MHz, PCLK2 = 72 MHz) z wykorzystaniem HSE jako źródłowego sygnału SYSCLK (patrz Rys. 1.23).

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

    RCC_HCLKConfig(RCC_SYSCLK_Div1);                         // HCLK=SYSCLK/1
    RCC_PCLK2Config(RCC_HCLK_Div1);                          // PCLK2=HCLK/1
    RCC_PCLK1Config(RCC_HCLK_Div2);                          // PCLK1=HCLK/2
    RCC_PLLConfig(RCC_PLLSource_HSE_Div1, RCC_PLLMul_9); // PLLCLK = (HSE/1)*9
                                                             // czyli 8MHz * 9 = 72 MHz
    RCC_PLLCmd(ENABLE);                                      // wlacz PLL
    while(RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET);  // czekaj na uruchomienie PLL
    RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);               // ustaw PLL jako zrodlo
                                                             // sygnalu zegarowego
    while(RCC_GetSYSCLKSource() != 0x08);                    // czekaj az PLL bedzie
                                                             // sygnalem zegarowym systemu
    // konfiguracja sygnalow taktujacych uzywanych peryferii
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);// wlacz taktowanie portu GPIO B
  }
}
```

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 peryferiali nie została opisana w

| 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 1 s przy zadanym zegarze HCLK

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.

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:

```
void GPIO_Config(void) {
  // konfigurowanie portow GPIO
  GPIO_InitTypeDef  GPIO_InitStructure;

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

Częstotliwość zmiany określa prędkość narastania sygnału wraz z jego zmianą – w przypadkach gdy nie jest to niezbędne, warto wybierać najniższą dozwoloną wartość. Wyjście w trybie *push-pull* oznacza, że sygnał wyjściowy przyjmuje wyłącznie dwie wartości – logiczne 0 i logiczne 1. 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:

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

oraz

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

Widoczna funkcja `GPIO_WriteBit` służy do nadawania wartości poszczególnym bitom portów wyjściowych. W tym przypadku korzystamy z portu B, na którym modyfikujemy wartość bitu 8, któremu odpowiada dioda o numerze 1. `Bit_SET` oraz `Bit_RESET` oznaczają odpowiednio ustawienie 1 i 0 logicznego (tj. odpowiednio zapalenie i zgaszenie diody).

Ostatnią funkcją jest, wspomniane wcześniej, programowe opóźnienie:

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

wykonuje ono w pętli: sprawdzenie warunku, dekrementację zmiennej oraz dwie puste instrukcje mikroprocesora *No Operation* (NOP). Otrzymane w ten sposób opóźnienie nie jest dokładne i wymaga wyłączenia optymalizacji kompilatora (inaczej może pominąć wykonanie takiego „bezużytecznego” kodu), lecz jest ono wystarczające do wstępnych testów. 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 1.2.6). Aby więc osiągnąć opóźnienie o długości 1 s przy zegarze HCLK o częstotliwości 72 MHz należy do funkcji `Delay` przekazać wartość 1535000.

**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);// wlacz taktowanie portu GPIO A
2
```

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
2
```

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; // wejscie 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 zwierał podany pin do masy w momencie jego wciśnięcia. Gdy przycisk nie jest wciśnięty, zwiera on podany pin przez rezystor do zasilania (3,3 V). Warto jednak zauważyć, że takie rozwiązanie w naszym przypadku jest redundantne. Dokładnie ten sam mechanizm został zrealizowany na płytce rozwojowej, co powoduje, że niejako użyte zostały dwa pociągnięcia w górę, co nie daje absolutnie żadnego zysku w stosunku do jednokrotnego podciągnięcia. Wynika z tego, że możemy wyłączyć podciągnięcie w górę w mikrokontrolerze stosując zamiast `GPIO_Mode_IPU` wartość `GPIO_Mode_IN_FLOATING`, co oznacza wyłączenie zarówno podciągania w górę jak i w dół.

Program ma działać tak, aby 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.

**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 *Drivers1602* 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 (zakończonego znakiem `'\0'`),
— `LCD_GoTo` – funkcja służąca do ustawienia kursora na zadaną pozycję.

Dokumentacja do wyświetlacza opisuje dokładnie poszczególne komendy, które są obsługiwane przez jego sterownik. Naśladując procedurę inicjalizacji, można wywołać przykładowo komendę przesunięcia **kursora** w **prawą** stronę o jedno miejsce:

| nazwa pinu | | we/wy | opis |
|---|---|---|---|
| LCD | STM32 | | |
| `RS` | PC12 | wy | *Register Select*, wybór rejestru: 0 – instrukcji, 1 – danych |
| `R/W` | PC11 | wy | *Read/Write*, kierunek transferu 0 – zapis, 1 – odczyt |
| `E` | PC10 | wy | *Enable*, sygnał zapisu/odczytu – aktywne zbocze opadające |
| `DB7` | PC0 | we/wy | *Data Bits: b4-7*, trój-stanowe wejścia/wyjścia danych. W trybie z transferem 4-bitowym te cztery bity wykorzystywane są do przesyłu dwóch połówek (*nibble*) bajtu danych |
| `DB6` | PC1 | we/wy | |
| `DB5` | PC2 | we/wy | |
| `DB4` | PC3 | we/wy | |
| `DB3` | — | — | *Data Bits: b0-3*, 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

```
LCD_WriteCommand(HD44780_DISPLAY_CURSOR_SHIFT |
                 HD44780_SHIFT_CURSOR |
                 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`).

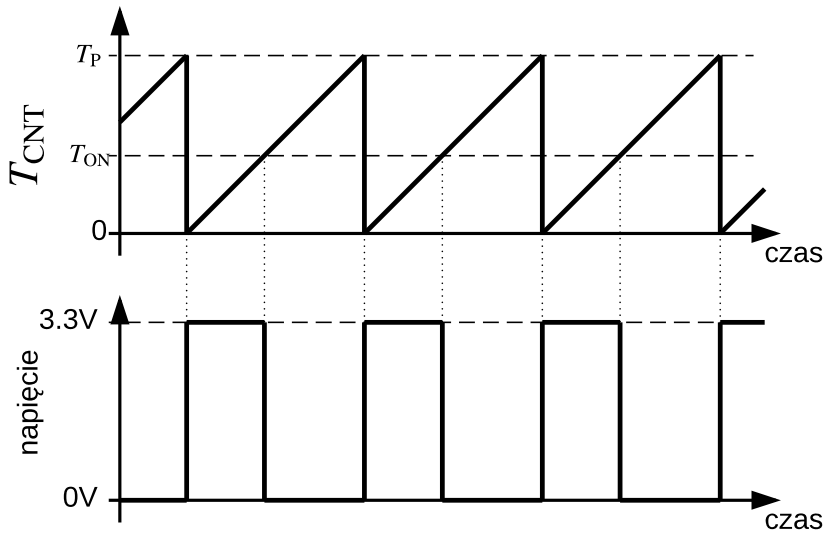
**Konfiguracja PWM:** Ponieważ mikrokontrolery z rodziny STM32 są wysoce konfigurowalne, poniższy opis ograniczy się do omówienia wyłącznie konfiguracji układu timera w trybie generacji sygnału PWM. Aby wykorzystywać timery w tworzonym programie należy dodać je do konfiguracji poprzez odkomentowanie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

```
#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 $T_{\mathrm{ON}}$ czasu utrzymywany jest stan wysoki, a przez $T_{\mathrm{OFF}}$ utrzymywany jest stan niski. $T_{\mathrm{ON}} + T_{\mathrm{OFF}} = T_{\mathrm{P}}$, gdzie $T_{\mathrm{P}}$ to czas trwania pojedynczego okresu. Szerokość wspomnianego impulsu może być wyrażona w procentach jako stosunek trwania sygnału wysokiego do okresu, tj. $\frac{T_{\mathrm{ON}}}{T_{\mathrm{P}}} \cdot 100\%$. W mikrokontrolerach osiągane jest to przy użyciu timera, który zlicza kolejne takty zegara (źródło zegara można skonfigurować stosownie do potrzeb) i porównuje wartość licznika $T_{\mathrm{CNT}}$ z wartością $T_{\mathrm{ON}}$. Jeśli wartość licznika jest mniejsza, to na wyjściu jest stan wysoki, jeśli jest większa, to stan niski. Przekroczenie wartości $T_{\mathrm{P}}$ powoduje automatyczne zresetowanie licznika (zakładamy zliczanie w górę). Na rys.




Rys. 1.24. Uproszczony wykres zależności między zawartością licznika $T_{\mathrm{CNT}}$, a postacią wygenerowanej fali PWM

1.24 widoczne jest (w uproszczeniu) jak generowana jest fala PWM. W dalszej części poszczególne wartości będą wynosić: $T_{\mathrm{ON}} = 1024$, $T_{\mathrm{OFF}} = 3071$, $T_{\mathrm{P}} = 4095$.

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

Konfiguracja pinu w trybie PWM została przedstawiona poniżej i zrealizowana na pinie PB8, do którego podłączony jest kanał 3 timera *TIM4* (zgodnie z tabelą 5: *Medium-density STM32F103xx pin definition*, z dokumentacji technicznej używanego mikrokontrolera – Rys. 1.25).

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; // szybkosc 50MHz
3  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;    // wyjscie w trybie alt. push-pull
4  GPIO_Init(GPIOB, &GPIO_InitStructure);
5
```

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  // konfiguracja timera
2  timerInitStructure.TIM_Prescaler = 0;              // prescaler = 0
3  timerInitStructure.TIM_CounterMode = TIM_CounterMode_Up; // zliczanie w gore
4  timerInitStructure.TIM_Period = 4095;              // okres dlugosci 4095+1
```

| 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. 1.25. Fragment noty katalogowej mikrokontrolera STM32F100, tabela 4. – rozpiska funkcji pinów

```
  timerInitStructure.TIM_ClockDivision = TIM_CKD_DIV1; // dzielnik czestotliwosci = 1
  timerInitStructure.TIM_RepetitionCounter = 0;      // brak powtorzen
  TIM_TimeBaseInit(TIM4, &timerInitStructure);       // inicjalizacja timera TIM4
  TIM_Cmd(TIM4, ENABLE);                             // aktywacja timera TIM4

  // konfiguracja kanalu timera
  outputChannelInit.TIM_OCMode = TIM_OCMode_PWM1;    // tryb PWM1
  outputChannelInit.TIM_Pulse = 1024;                // wypelnienie 1024/4095*100% = 25%
  outputChannelInit.TIM_OutputState = TIM_OutputState_Enable; // stan Enable
  outputChannelInit.TIM_OCPolarity = TIM_OCPolarity_High; // polaryzacja Active High
  TIM_OC3Init(TIM4, &outputChannelInit);             // inicjalizacja kanalu 3 timera TIM4
  TIM_OC3PreloadConfig(TIM4, TIM_OCPreload_Enable); // konfiguracja preload register
}

```

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

```
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4 , ENABLE);// wlacz 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:

```
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);// wlacz 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 $T_{\mathrm{ON}}$ do odpowiedniego rejestru mikrokontrolera:

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

```

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

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

```

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

**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`:

```
#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 konfigurującym 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),
— 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`):

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

można przejść do implementacji opisanej konfiguracji:

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

  ADC_DeInit(ADC1);                                    // reset ustawien ADC1

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;            // pin 4
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;    // szybkosc 50MHz
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;// wyjscie w floating
  GPIO_Init(GPIOC, &GPIO_InitStructure);

  ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;  // niezalezne dzialanie ADC 1 i 2
  ADC_InitStructure.ADC_ScanConvMode = DISABLE;       // pomiar pojedynczego kanalu
  ADC_InitStructure.ADC_ContinuousConvMode = DISABLE; // pomiar na zadanie
  ADC_InitStructure.ADC_ExternalTrigConv=ADC_ExternalTrigConv_None; // programowy start
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right; // pomiar wyrownany do prawej
  ADC_InitStructure.ADC_NbrOfChannel = 1;             // jeden kanal
  ADC_Init(ADC1, &ADC_InitStructure);                 // inicjalizacja ADC1
  ADC_RegularChannelConfig(ADC1, 14, 1, ADC_SampleTime_1Cycles5); // ADC1, kanal 14,
                                                        // 1.5 cyklu
  ADC_Cmd(ADC1, ENABLE);                              // aktywacja ADC1

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

```

Jak widać pin służący do pomiaru analogowej wartości napięcia został skonfigurowany jako 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. 1.26. 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 MHz / 6 = 12 MHz) – tak też konfigurujemy ten zegar. W tym celu dodajemy do funkcji `RCC_Config` następującą linijkę:

```
1 RCC_ADCCLKConfig(RCC_PCLK2_Div6);                    // ADCCLK = PCLK2/6 = 12 MHz
2
```

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.

| 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. 1.26. Fragment noty katalogowej mikrokontrolera STM32F100, tabela 4. – rozpiska funkcji pinów

## 1.3. Wykonanie ćwiczenia

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

1. implementacja programu na płytkę 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,
2. implementacja programu zapalającego diodę podłączoną do pinu PB9 pod wpływem wciśnięcia przycisku SW0,
3. implementacja programu wyświetlającego na wyświetlaczu LCD, w dwóch linijkach, 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).