Rysunek 1: Zestaw uruchomieniowy ZL27ARM

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

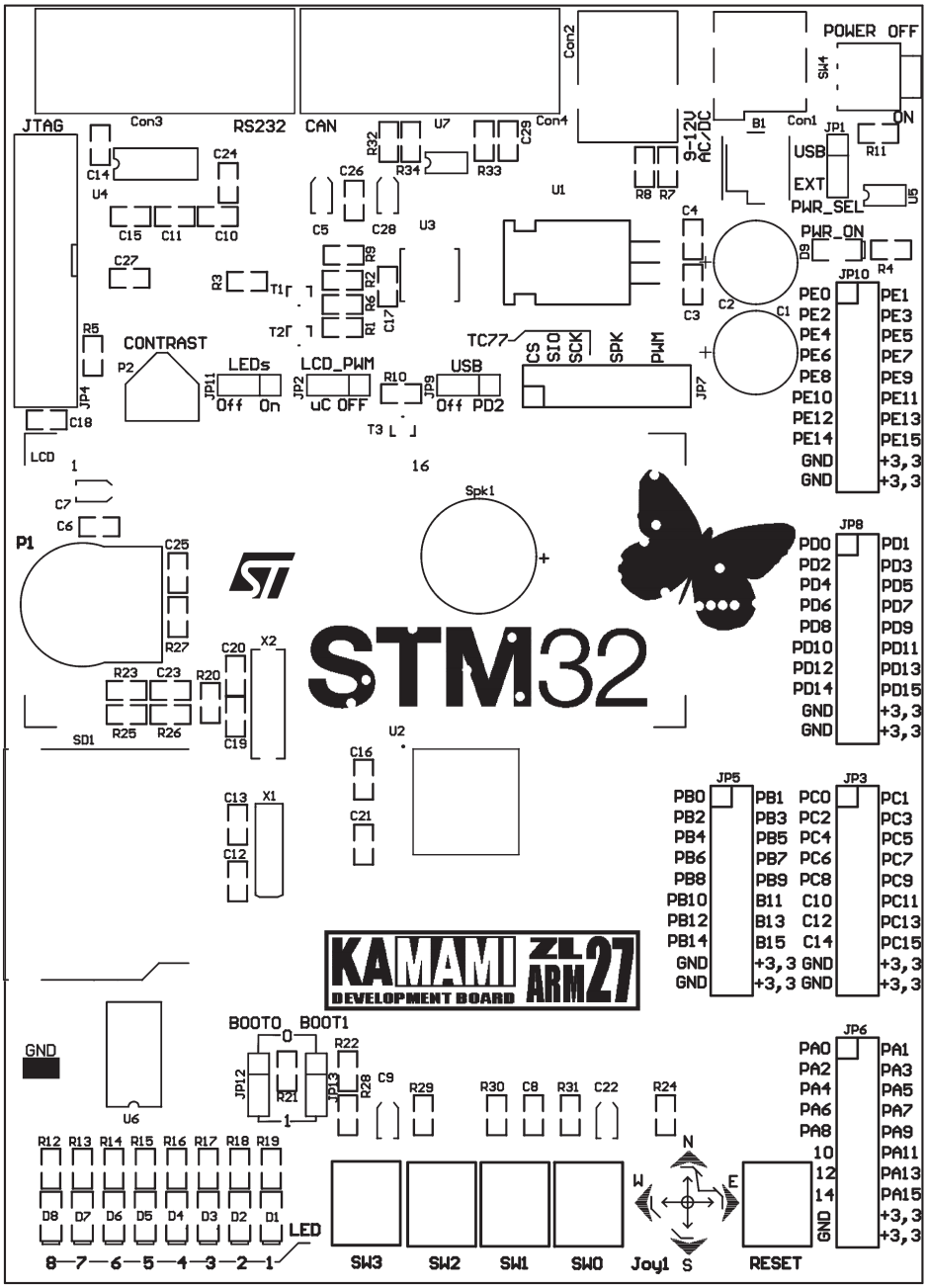
1.1 Cel

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

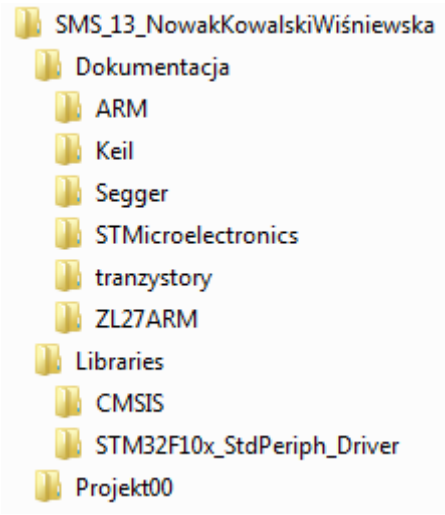
1.2 Przebieg laboratorium (prowadzenie za rączkę)

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

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



Rysunek 2: Schemat zestawu uruchomieniowy ZL27ARM



Rysunek 3: Struktura katalogów dla projektu 00

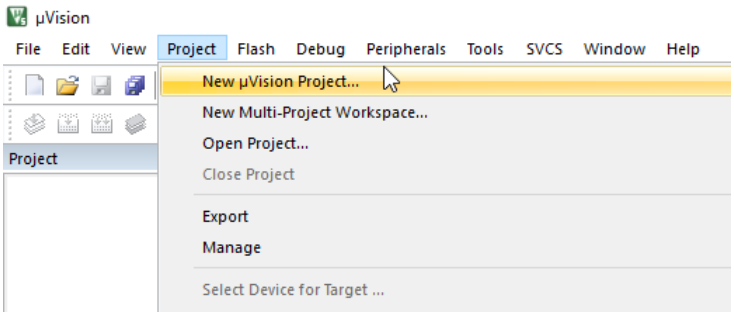
STM32F103VB (All Devices → STMicroelectronics → STM32F1 Series → STM32F103 → STM32F103VB). Po prawej stronie zaktualizuje się lista paczek Zainstalować paczkę `Keil::STM32F1xx_DFP` i zaktualizować paczkę `ARM::CMSIS`. Tak przygotowane środowisko pozwoli na symulację wyżej wspomnianego mikrokontrolera,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

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

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

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

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



Rysunek 4: Tworzenie nowego projektu

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

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

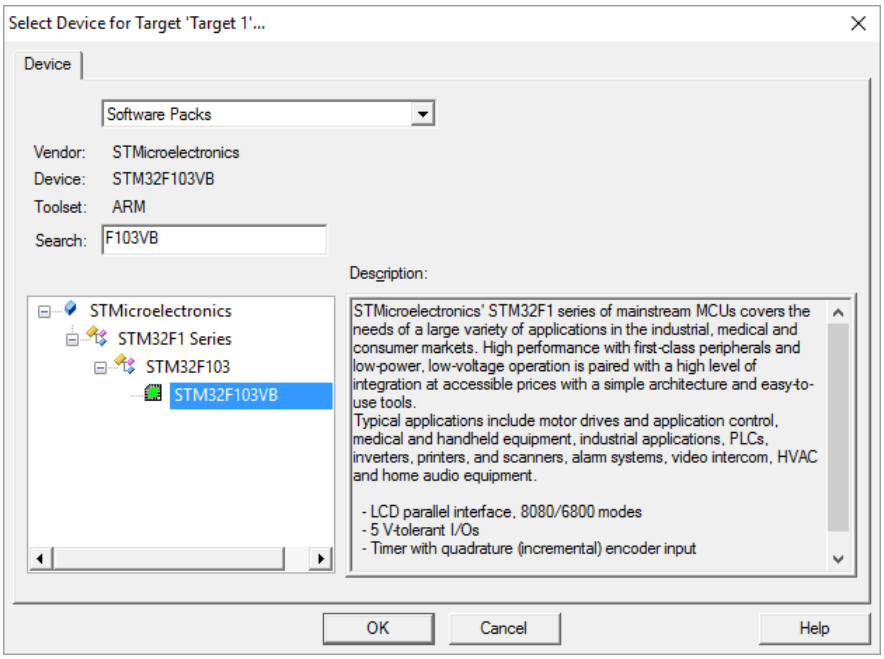
Inicjalizację projektu należy rozpocząć od stosownego podziału plików na katalogi. W tym celu w katalogu głównym `Target 1` należy utworzyć katalogi (kliknąć prawym przyciskiem myszy na `Target 1` i wybrać opcję `Add Group...`): `UserCode` (na kod użytkownika), `StdPeriphDrv` (na biblioteki do obsługi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

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

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

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

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



Rysunek 5: Wybór mikrokontrolera

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

- do `CMSIS` dodać:

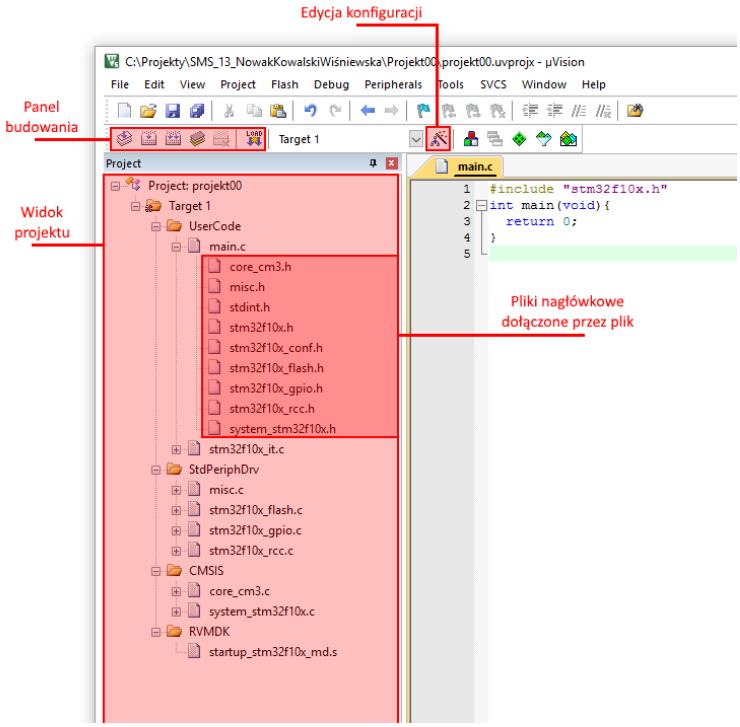
  - `..\Libraries\CMSIS\CM3\CoreSupport\core_cm3.c`
  - `..\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\system_stm32f10x.c`

- do `RVMDK` dodać:

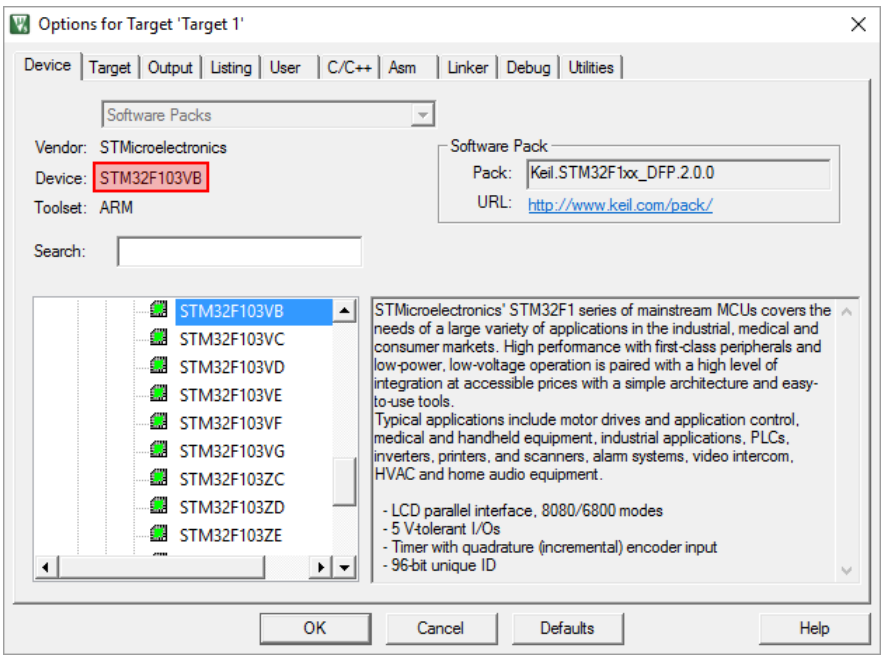
  - `..\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\arm↩`
    `↪ \startup_stm32f10x_md.s`

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

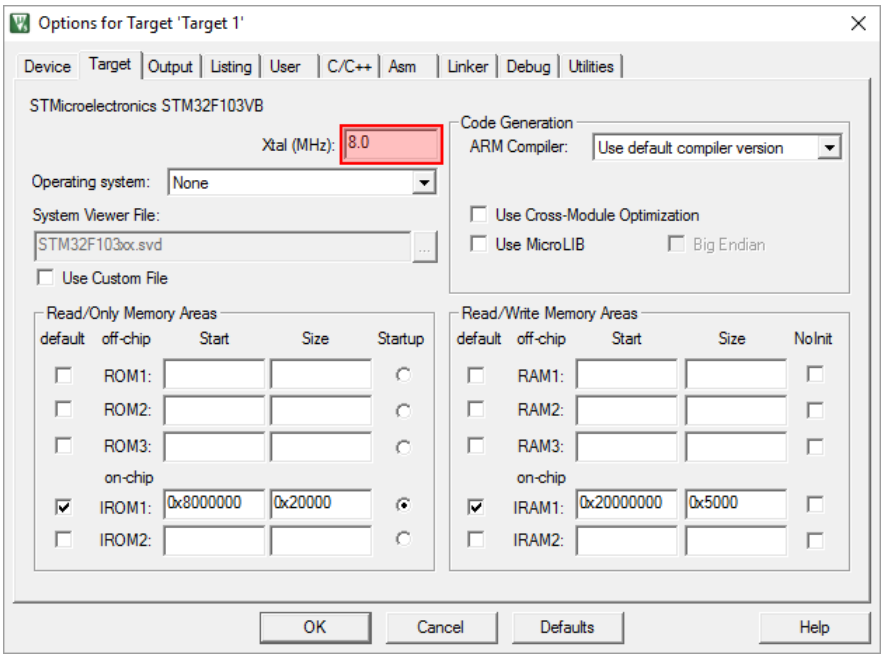
Gdy już wszystkie pliki są dodane do projektu można przejść do jego konfiguracji. W tym celu należy dwukrotnie kliknąć *Target 1* z drzewa projektu, a następnie wybrać menu *Project* → *Options for Target 'Target 1'...*. Konfiguracja powinna uwzględniać:



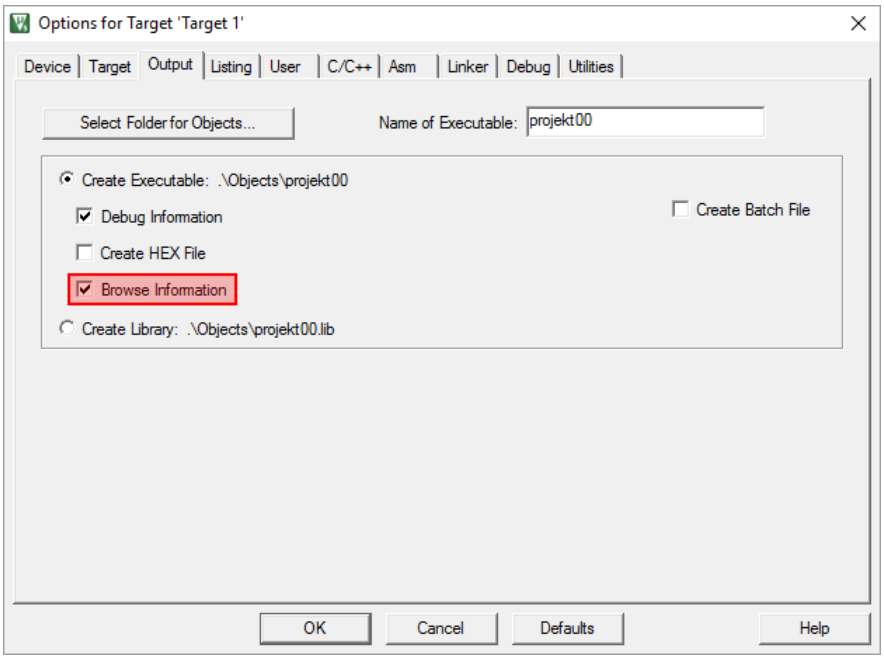
Rysunek 6: Widok projektu z gotową strukturą plików



Rysunek 7: Konfiguracja projektu – zakładka Device

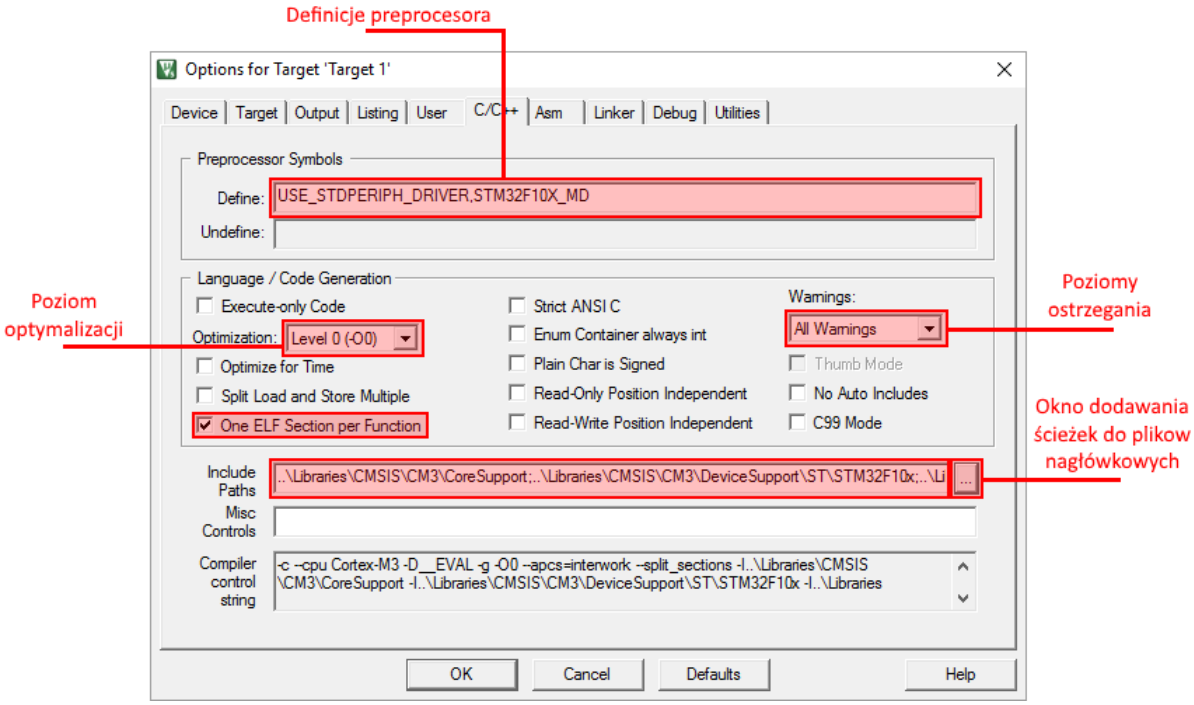


Rysunek 8: Konfiguracja projektu – zakładka Target

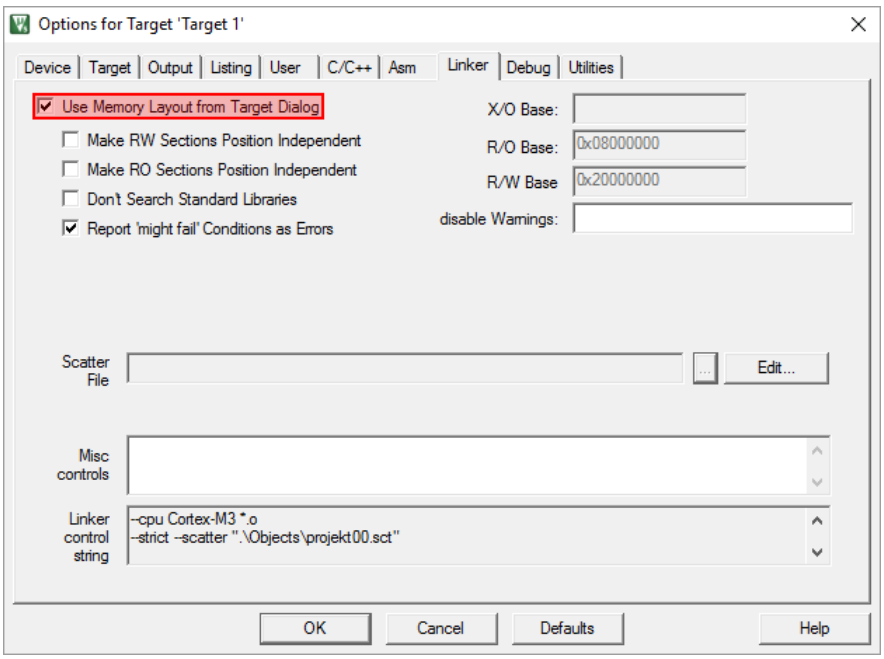


Rysunek 9: Konfiguracja projektu – zakładka Output

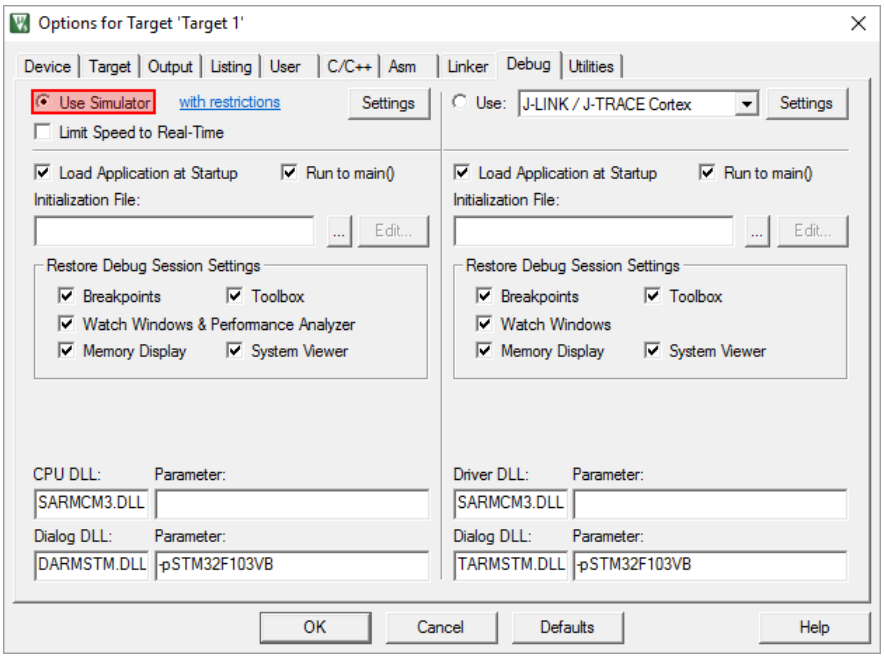
- w zakładce Device (Rys. 7)
  - wybór odpowiedniego mikrokontrolera, tj. STM32F103VB,
- w zakładce Target (Rys. 8)
  - ustawienie odpowiedniej częstotliwości kwarcu *Xtal (MHz)*, tj. na 8,0 (taki właśnie jest zamontowany w rozważanym zestawie uruchomieniowym),
- w zakładce Output (Rys. 9)
  - zaznaczenie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



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



Rysunek 11: Konfiguracja projektu – zakładka Linker



Rysunek 12: Konfiguracja projektu – zakładka Debug

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

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

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

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

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

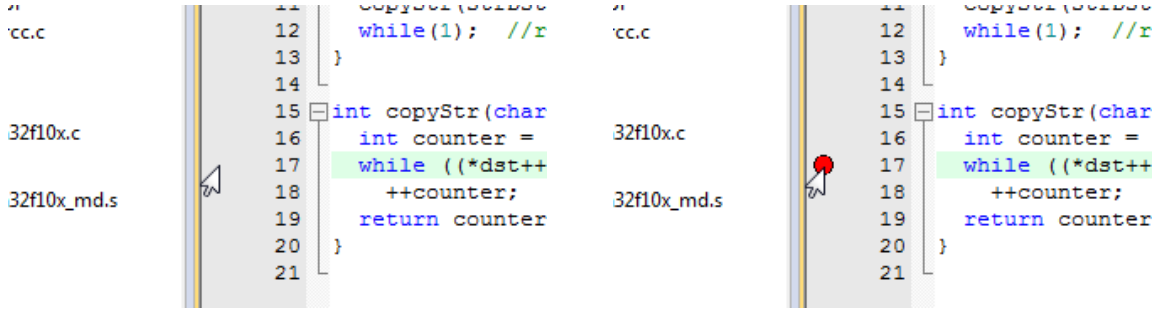
```
/**************************************
 * projekt00: symulacja komputerowa   *
 **************************************/
#include "stm32f10x.h"

char strDst[32] = "\0";

int copyStr(char *, char *);

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

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

a następnie ponownie skompilować cały projekt.

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

(a) (b) (c) (d) (e)

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



(a) (b)

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

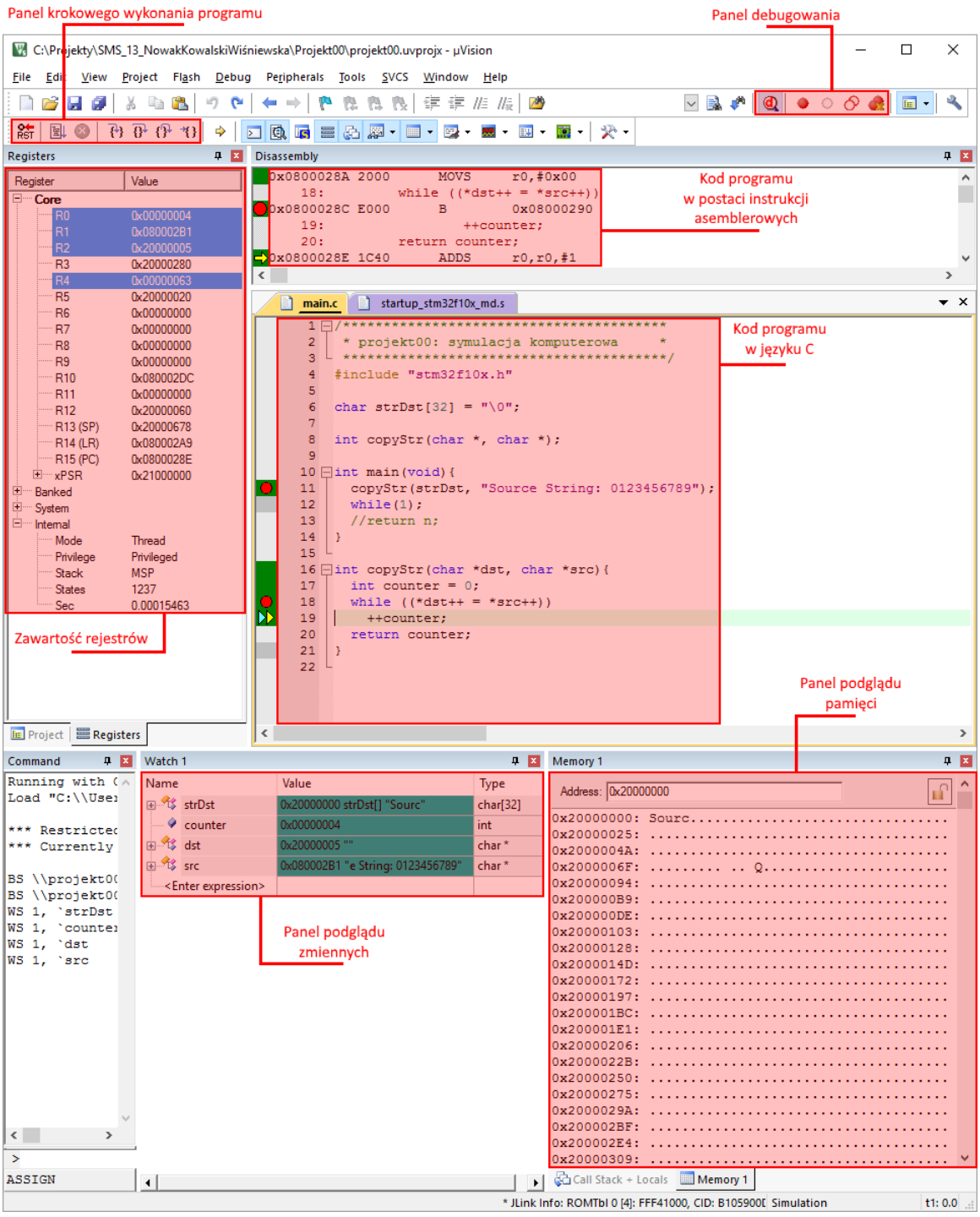
użytkownik chce wstrzymać działanie programu. Należy jednak pamiętać, że pojedyncza linia kodu może zostać skompilowana do kilku instrukcji asemblerowych, a w przypadku włączenia optymalizacji mogą być widoczne uproszczenia kodu utrudniające porównanie kodu w języku C oraz kod asemblerowy.

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

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

(a) (b) (c) (d) (e) (f) (g) (h)

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



Rysunek 16: Widok debugowania

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

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

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

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

Jako mikrokontroler użyty zostanie STM32F103VB, zawierający się w zestawie uruchomieniowym

ZL27ARM. Ogólne informacje dotyczące tego zestawu znajdują się w pierwszych rozdziałach, natomiast szczegółowe informacje na temat samego mikrokontrolera STM32F103VB można znaleźć w dokumentacji znajdującej się na stronie producenta i w katalogu `Dokumentacja` (wartymi uwagi są pliki Datasheet oraz RM0008). Programowanie zestawu ZL27ARM (Rys. 1) odbywać się będzie za pośrednictwem programatora *j-link* (firmy *Segger*) w wersji edukacyjnej (*EDU*) – Rys. 17. Jest on podłączany poprzez złącze JTAG (*Joint Test Action Group*), które pozwala na testowanie (w tym debugowanie i śledzenie wykonania programu) procesora wlutowanego w zmontowaną płytę drukowaną. Połaczenie 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 podłaczenie programatora powinno być sygnalizowane przez świecenie się (z okresowym chwilowym przygasaniem) zielonej diody znajdującej sie na jego obudowie, nad logo producenta.

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

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

Przekłada się to na następujące ustawienia zworek:

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 zasilania *POWER* ustawionym na *OFF*) do złącza opisanego etykietą *Con2* należy podłączyć wtyczkę typu B, natomiast do komputera wtyczkę typu A. Po podłączeniu wszystkich elementów można ustawić przełącznik zasilania *POWER* w położenie *ON*. Jeśli wszystko zostało zrealizowane poprawnie powinna zapalić się zielona LED o nazwie *PWR_ON*.

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

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

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

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

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

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

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

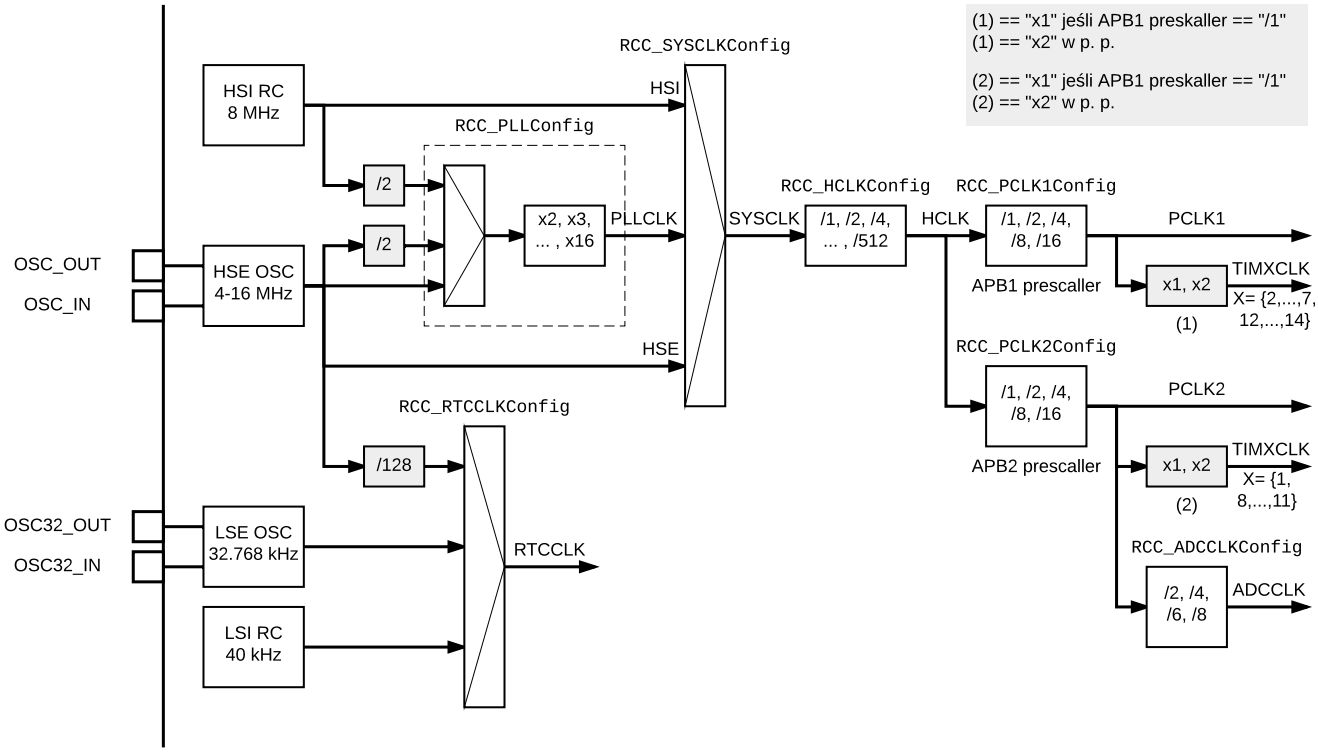
1.3 Przebieg laboratorium (samodzielnie wykonywane zadanie)

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

- `FLASH_Latency_0` jeśli $0 < \text{SYSCLK} \leqslant 24\,\text{MHz}$
- `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 zegaru 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`):

```c
/**************************************
 * projekt01: konfiguracja zegarow    *
 **************************************/
#include "stm32f10x.h"
#include <stdbool.h>   // true, false

#define DELAY_TIME 8000000

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

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

  while(1) {            // 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 poniższą tabelą

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

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

Diody w poprzednich ćwiczeniach były wyłączone poprzez ustawienie zworki `JP11` o nazwie `LEDs` na `Off`. Aby można było je kontrolować należy wyłączyć mikrokontroler, przestawić zworkę na pozycję `On` i ponownie włączyć mikrokontroler. Diody najprawdopodobniej rozświetlą się z czasem mimo braku jakiejkolwiek interakcji ze strony użytkownika. Jest to ciekawe zjawisko wynikające z niepodciągnięcia wyjść prowadzacych 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. 18).

```
bool 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
    return true;
  }
  return false;
}
```

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 (należy prawym przyciskiem myszy kliknąć na nazwę funkcji i wybrać `Go To Definition Of <nazwa funkcji>`). Niestety standardowa biblioteka peryferiali 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.

Na koniec inicjalizacji warto także włączyć taktowanie peryferiali – w tym ćwiczeniu potrzebna jest wyłącznie jedna dioda znajdująca się na płycie uruchomieniowej, podłączona do pinu 8 portu B. Konfiguracja tego pinu znajduje sie 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. Jak okaże się w późniejszych ćwiczeniach nie jest to jedyny wybór do dyspozycji – na potrzeby tego ćwiczenia jest on jednak najrozsądniejszy.

Funkcje służące do obsługi diody LED są zdefiniowane następująco:

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

```
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 programowe opóźnienie:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

nazwa pinu		we/wy	opis
LCD	STM32		
`RS`	PC12	wy	*Register Select*, wybór rejestru: 0 – rejestr instrukcji, 1 – rejestr 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*, trzystanowe 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*, trzystanowe wejścia/wyjścia danych. W trybie z transferem 4-bitowym te cztery bity są niewykorzystywane
`DB2`	—	—	
`DB1`	—	—	
`DB0`	—	—	

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

Najważniejszymi funkcjami dostępnymi w ramach biblioteki są:

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

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

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

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

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

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

- LED,
- przycisk,
- potencjometr.

Z wykorzystaniem tej płytki należy napisać program, który będzie:

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

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

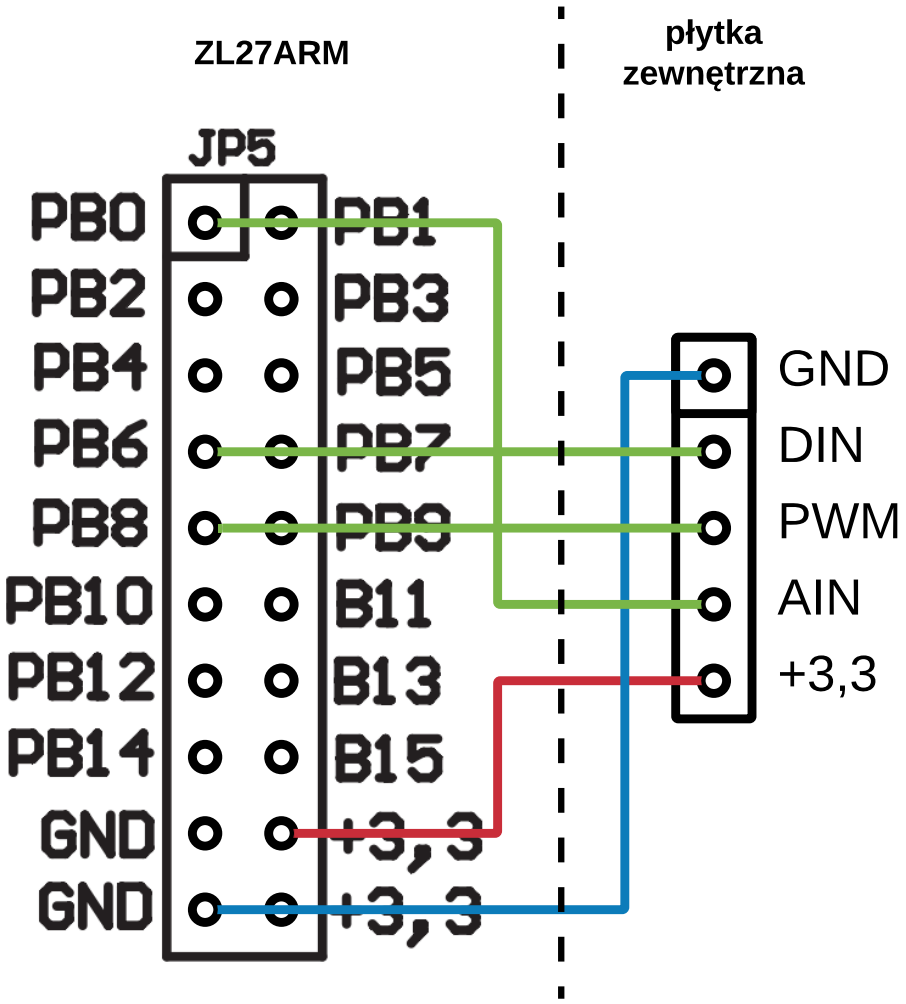
```
#include "stm32f10x_tim.h"
#include "stm32f10x_adc.h"
```

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

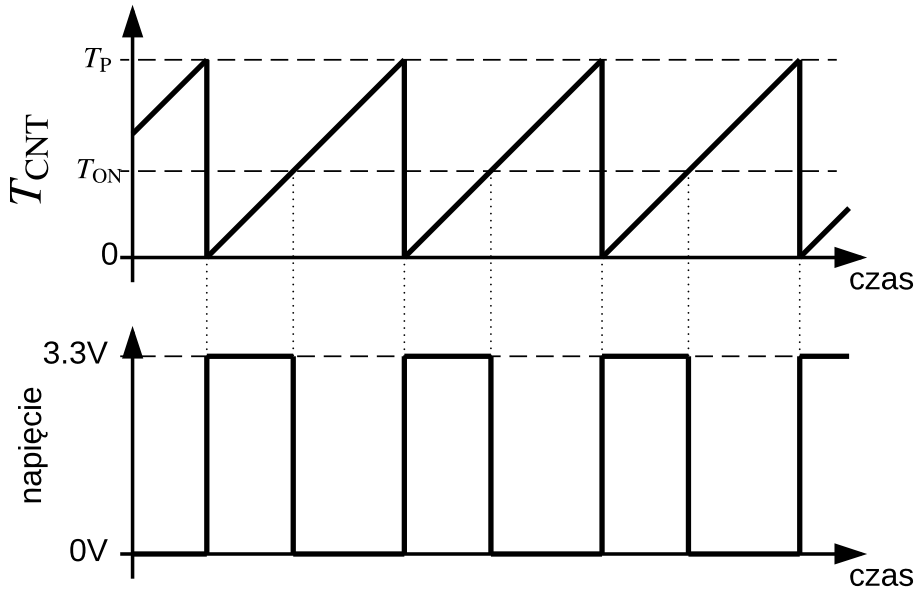
- do PB0 – potencjometr, pozwalający na odczyt napięcia z zakresu od 0 V do 3,3 V,
- do PB6 – przycisk pozwalający na odczyt napięcia 0 V do 3,3 V (lub innej wartości jeśli przycisk nie będzie podciągnięty),
- do PB8 – LED, którego jasnością świecenia będzie można sterować poprzez zmianę wypełnienia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

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

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



Rysunek 19: Schemat podłączenia płytki zewnętrznej do płytki uruchomieniowej

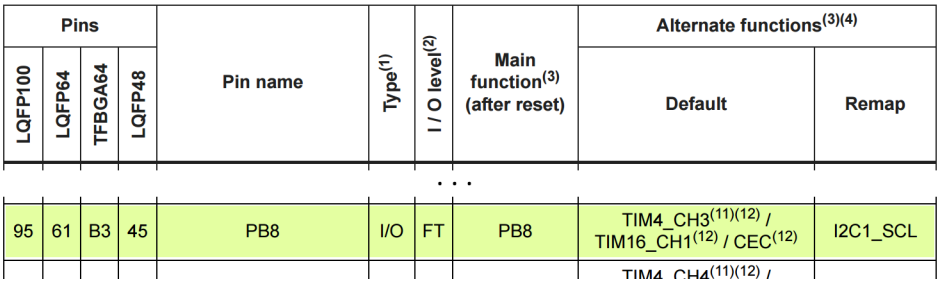


Rysunek 20: Uproszczony wykres zależności między zawartością licznika T_{CNT}, a postacią wygenerowanej fali PWM

sterowanie szerokością impulsu. Realizowane jest to poprzez okresową zmianę wartości logicznej na wyjściu jednego z pinów, w taki sposób, że przez T_{ON} czasu utrzymywany jest stan wysoki, a przez T_{OFF} utrzymywany jest stan niski. $T_{\mathrm{ON}} + T_{\mathrm{OFF}} = T_{\mathrm{P}}$, gdzie T_{P} to czas trwania pojedyczego 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_{CNT} z wartością T_{ON}. Jeśli wartość licznika jest mniejsza, to na wyjściu jest stan wysoki, jeśli jest większa, to stan niski. Przekroczenie wartości T_{P} powoduje automatyczne zresetowanie licznika (zakładamy zliczanie w górę). Na rys. 20 widoczne jest (w uproszczeniu) jak generowana jest fala PWM. W dalszej części poszczególne wartości będą wynosić: $T_{\mathrm{ON}} = 1024$, $T_{\mathrm{OFF}} = 3071$, $T_{\mathrm{P}} = 4095$.

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. 21).



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

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

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

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

Konfiguracja PWM:

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

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

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

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

```
  outputChannelInit.TIM_Pulse = 1024;                // 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_{ON} do odpowiedniego rejestru mikrokontrolera:

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

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

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

Mikrokontrolery bardzo często wyposażone są w przetworniki analogowo-cyfrowe. Dzięki nim napięcie przyłożone do pinu wejściowego może zostać odczytane jako wartość cyfrowa. W przypadku mikrokontrolera zawartego na płytce uruchomieniowej ZL27ARM do dyspozycji są 2 12-bitowe przetworniki analogowo-cyfrowe (do 16 kanałów każdy). Przetworniki te mierzą napięcia w zakresie od 0 V do 3,3 V. Oznacza to jednocześnie, że sygnał o maksymalnej wartości napięcia (3,3 V) zostanie zinterpretowany jako wartość `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 8) przetwornika ADC1,
- start pomiaru rozpoczyna się na programowe żądanie użytkownika,
- pomiar trwać będzie możliwie krótko (tutaj 1,5 cyklu + stały czas przetwarzania 12,5 cyklu – 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_0;               // pin 0
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;       // szybkosc 50MHz
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;// wyjscie w floating
    GPIO_Init(GPIOB, &GPIO_InitStructure);

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

    ADC_ResetCalibration(ADC1);                         // reset 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 PB0 będzie podłączony do kanału 8 przetwornika analogowo cyfrowego ADC1?". Odpowiedź na to pytanie wymaga przestudiowania noty katalogowej mikrokontrolera STM32F100, a dokładniej tabeli 4, gdzie można znaleźć wpis widoczny na Rys. 22. Należy zauważyć, że mimo, że podłączenie do ADC jest funkcją alternatywną, sam pin jest skonfigurowany jako pin wejściowy – nie jest to reguła koniecznie stosowana w innych mikrokontrolerach, nawet tych z rodziny STM32.

Na końcu przedstawionego kodu widoczna jest procedura kalibracji przetwornika. Należy (choć nie jest to konieczne) ją przeprowadzić w celu osiągnięcia dokładniejszych pomiarów. Przed uruchomieniem przetwornika należy pamiętać o włączeniu jego zegara, poprzedzając to odpowiednią konfiguracją prescallera

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

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

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

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

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

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

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

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

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

Podsumowując, wszystkie instrukcje wykonywane były jedna po drugiej, co powodowało utrudnioną interakcję z mikrokontrolerem. Oczekiwanym rezultatem byłoby aby wciśnięcie przycisku powodowało natych-

miastową reakcję mikrokontrolera (co jak się okaże również nie jest tak skuteczne jak mogłoby się zdawać), natomiast część akcji powinna być wykonywana „w tle”(np. wykorzystanie przetwornika analogowo-cyfrowego). To właśnie zostanie zaimplementowane poprzez wykorzystanie liczników i timerów.