

# Programowanie mikrokontrolerów STM32

Projekt „Matematyka dla Ciekawych Świata”,  
Krzysztof Lasocki <krz.lasocki@gmail.com>,  
Robert Ryszard Paciorek <rrp@opcode.eu.org>

2023-06-25

Skrypt opisuje podstawy programowania mikrokontrolerów STM32. Podczas kursu będziemy używać popularnej, dostępnej i prostej płytki “Blue Pill”. Programy będą pisane w języku C z pomocą biblioteki libopencm3 (<https://github.com/libopencm3/libopencm3>). Każde z ćwiczeń to oddzielny program, więc jego kod znajduje się w oddzielnym katalogu.

Kod przykładów znajduje się w repozytorium git. Można je sklonować do podkatalogu za pomocą:

```
git clone https://bitbucket.org/OpCode-eu-org/stm32-examples.git
```

Po sklonowaniu repozytorium w jego katalogu głównym należy utworzyć link o nazwie libopencm3 wskazujący na katalog w którym znajduje się (skompilowana wcześniej) biblioteka:

```
cd stm32-examples  
ln -s SCIEŻKA/DO/libopencm3 libopencm3
```

Jeżeli nie masz jeszcze pobranej i skompilowanej biblioteki libopencm3, to zamiast tworzyć link możesz umieścić tutaj katalog z tą biblioteką. Ważne jest zachowanie nazwy i umieszczenie go w głównym katalogu repozytorium z przykładami, gdyż pliki wspomagające kompilację przykładów zakładają że w tym miejscu znajdują katalog z tą biblioteką.

## Ważne

Każdy z przykładowych programów znajduje się w swoim własnym katalogu. Katalog taki zawiera wszystkie pliki źródłowe programu, plik Makefile i będzie też zawierać pliki tymczasowe i wynikowe powstałe w wyniku kompilacji. Do kompilacji i wgrywanie programów do mikrokontrolera będziemy korzystać z narzędzia make, które z odpowiednimi argumentami należy wywołać w katalogu danego przykładu. Typowo będziemy używać 3 wywołań programu make:

- `make` – skompiluje i zlinkuje z potrzebnymi fragmentami biblioteki nasz program
- `make install` – skompiluje i zlinkuje z potrzebnymi fragmentami biblioteki nasz program oraz wgra go do mikrokontrolera
- `make run` – uruchomi program znajdujący się w mikrokontrolerze po jego restarcie (jest to szybsze i bardziej przyjazne dla pamięci mikrokontrolera niż używanie `make install` z tym samym kodem)

Make jest narzędziem automatyzującym proces kompilacji oprogramowania, wykorzystującym pliki Makefile. Więcej na jego temat można znaleźć w skrypcie [narzędzia deweloperskie](#). Pomimo iż nasze programy są małe to używamy go aby ułatwić sobie kompilację kodu dla mikrokontrolera - poprzez zastosowanie odpowiedniego kompilatora (który wygeneruje kod dla procesora ARM, a nie dla x86 pod kontrolą którego działa), przekazanie do niego opcji ustawiających jaki mikrokontroler używany oraz dolinkowanie odpowiednich bibliotek.

# 1 Przygotowanie

## 1.1 Instalacja i przygotowanie narzędzi

Do programowania mikrokontrolera STM32 użyjemy następujących narzędzi programistycznych:

- Toolchain<sup>1</sup>dla architektury *arm-none-eabi* (`gcc-arm-none-eabi` oraz `binutils-arm-none-eabi`)
- Implementację libC (`libstdc++-arm-none-eabi-newlib`)
- Narzędzie do programowania przez UART (`stm32flash`)
- Program terminalowy do obsługi portów szeregowych (`picocom`)
- Bibliotekę `libopencm3`

Wszystkie narzędzia, oprócz ostatniego, dostępne są w repozytoriach Debiana (i systemów na nim opartych). Możesz je zainstalować za pomocą polecenia:

```
sudo apt install gcc-arm-none-eabi binutils-arm-none-eabi
↪ libstdc++-arm-none-eabi-newlib stm32flash picocom git make
```

Instalacja i kompilacja `libopencm3` odbywa się w następujący sposób:

```
git clone https://github.com/libopencm3/libopencm3.git
cd libopencm3
make
```

Pamiętaj o ustawieniu ścież

## 1.2 Połączenie mikrokontrolera

Aby móc uruchomić nasz kod na mikrokontrolerze, musimy podłączyć go do komputera w celu wgrywania na niego skompilowanych programów. STM32 można programować na wiele sposobów, my będziemy używać do tego interfejsu UART<sup>2</sup>.

Aby zaprogramować mikrokontroler musisz podłączyć go do przejściówki USB-UART (nie podłączaj jeszcze przejściówki do komputera). Za pomocą pasujących kabelków podłącz następujące piny przejściówki do pinów na mikrokontrolerze:

- masę (GND) do (dowolnej) masy mikrokontrolera (GND lub G)
- RX do TX mikrokontrolera (pin A9)
- TX do RX mikrokontrolera (pin A10)
- 5V do 5V mikrokontrolera

Za pomocą obciążków lub pęsety przełoż górna (patrzac na mikrokontroler tak aby port USB był po lewej) zworkę na pozycję "1".

Sprawdź wszystkie połączenia i podłącz przejściówkę do portu USB komputera. Na mikrokontrolerze powinna zaświecić się tylko czerwona dioda PWR.

Możesz sprawdzić połączenie z mikrokontrolerem używając `stm32flash` do wyświetlenia informacji na temat podłączonego mikrokontrolera. Jeżeli polecenie poniżej nie zgłosi błędów, oznacza to, że wszystko działa poprawnie.

```
stm32flash /dev/ttyUSB0
```

Tak przygotowany mikrokontroler jest gotowy do pracy.

1. Toolchain to zbiorowe określenie na narzędzia do przekształcania kodu źródłowego na kod maszynowy. Zawiera m.in. kompilator, linker (konsolidator), narzędzia obsługujące pliki obiektowe oraz kopujące dane między formatami plików wykonywalnych (`objcopy`) itd.
2. Inne z nich to SWD oraz JTAG

## 2 Pierwszy program

Odpowiednikiem programu “Hello, world!” w elektronice jest program migający diodą LED. Przykładowy kod takiego programu znajdziesz w pliku `main.c` znajdującym się w podkatalogu `10_blink` repozytorium z przykładami. Plik ten ma następującą postać:

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>

int main(){
    // Uruchomienie peryferium portu C
    // Włączenie sygnału zegara dla portu C
    rcc_periph_clock_enable(RCC_GPIOC);
    // Ustawienie pinu C13 w trybie wyjścia
    gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);

    while(1){
        // Poczekaj chwilkę
        for (int i = 0; i < 150000; i++) __asm__("nop");
        // Przełącz stan pinu 13 w porcie C
        gpio_toggle(GPIOC, GPIO13);
    }
}
```

Skompiluj i wgraj program za pomocą:

```
cd 10_blink
make install
```

Jeśli `stm32flash` wyszedł bez błędów, Twój program powinien się uruchomić. Zielona dioda na płytce powinna zacząć migać.

Jeżeli `stm32flash` zgłosił błędy zrestartuj mikrokontroler przyciskiem reset i ponów polecenie `make install`.

### Uwaga

Płytką BluePill której używamy posiada zworki kontrolujące tryb uruchomienia mikrokontrolera (piny `B00T0` i `B00T1`). Zwórka, którą przestawiłeś/aś (`B00T0`) na “1” podczas przygotowywania mikrokontrolera do pracy powoduje uruchomienie w trybie bootloadera. Po zaprogramowaniu, `stm32flash` wydaje bootloaderowi polecenie uruchomienia programu.

Jeśli chcesz, aby mikrokontroler automatycznie uruchamiał program po podłączeniu zasilania, możesz ustawić obie zworki na “0”. Nie zapomnij jednak ustawić ich z powrotem do trybu bootloadera, jeśli chcesz ponownie wgrać program.

Możesz użyć polecenia `make run` aby uruchomić program znajdujący się w pamięci mikrokontrolera po jego zrestartowaniu (bez potrzeby ponownego wgrywania lub zmieniania ustawień zwork).

Przyjrzyjmy się powyższemu plikowi linia po linii, aby zrozumieć, dlaczego nasz program działa.

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>
```

Dołączamy dwa pliki nagłówkowe z biblioteki libopencm3 aby móc używać jej funkcji. Możliwe jest pisanie kodu w “czystym” C (lub nawet w assemblerze), ale kod w ten sposób napisany będzie mniej czytelny i mniej przenośny.<sup>3</sup>

```
int main() {
```

Jak każdy program w C, funkcją początkową jest main. W tym przypadku nie bierze ona żadnych argumentów. Mimo int w definicji, nie zwraca ona żadnej wartości. W elektronice, main z reguły nigdy nie kończy pracy (powrót z niej najczęściej kończy się skokiem do wektora resetu i zresetowaniem mikronontrolera)

```
rcc_periph_clock_enable(RCC_GPIOC);  
gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);
```

Po rozpoczęciu programu konfigurujemy peryferia. W tym programie używamy portu C aby migać diodą, która jest podłączona do pinu C13 (13. bit portu C). Przed rozpoczęciem jakichkolwiek działań z tym peryferium musimy uruchomić jego zegar<sup>4</sup>(wywołaniem makra rcc\_periph\_clock\_enable z odpowiednim parametrem)

Następnie konfigurujemy pin C13 jako wyjście *push-pull*. Domyślnie wszystkie piny GPIO są skonfigurowane jako wejścia.

```
while(1){  
    // Poczekaj chwilkę  
    for (int i = 0; i < 150000; i++) __asm__("nop");  
    // Przełącz stan pinu 13 w porcie C  
    gpio_toggle(GPIOC, GPIO13);  
}
```

Jak mówiłem wcześniej, procedura main z reguły nie wychodzi. Zamiast tego kończy się nieskończoną pętlą. W pętli, procesor najpierw wykonuje nop, czyli tzw. pustą instrukcję 150000 razy<sup>5</sup>. Następnie funkcja gpio\_toggle zmienia stan pinu 13 w porcie C na przeciwny, co powoduje zapalenie lub zgaszenie LEDa.

### Zadanie 2.0.1

Zmień program 10\_blink tak, aby dioda LED migała około dwa razy wolniej.

## 3 Obsługa wejść

W tym ćwiczeniu pokażę, jak odczytywać stany logiczne pinów GPIO procesora oraz jak na ich podstawie podejmować decyzje. Skompiluj ten kod podobnie jak w poprzednim przykładzie (weź pod uwagę, że znajduje się on w innym katalogu: 11\_di). Plik main.c wygląda następująco:

3. Oraz niewiele szybszy.
4. Taka dowolność we włączaniu lub wyłączaniu sygnału zegara do peryferiów pozwala projektantom oszczędzać energię. Jest to bardzo ważne np. przy układach zasilanych bateriami. W układach CMOS, kiedy nie następują zmiany stanów, pobór energii jest praktycznie znikomy, więc to, czy sygnał zegara nieużywanego peryferium jest zatrzymany lub nie, znacznie wpływa na pobór prądu
5. Wbrew pozorom ta funkcja nie zabierze 150 tys. cykli procesora, tylko znacznie więcej. Zwiększenie wartości zmiennej, porównanie i skok warunkowy zajmują czas. Nie jest to precyzyjna metoda odmierzenia czasu. Użycie wstawki assemblerowej - bo to oznacza \_\_asm\_\_() z instrukcją nop - zapobiega optymalizacji takiej pętli przez kompilator. Wstawek tego typu można też używać w celu umieszczenia w kodzie C lub C++ dowolnych innych instrukcji assemblerowych.

```

#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>

int main(){
    // Uruchomienie peryferiów portów A, C
    // Włączenie sygnału zegara dla portów A, C
    rcc_periph_clock_enable(RCC_GPIOA);
    rcc_periph_clock_enable(RCC_GPIOC);

    // Ustawienie pinu C13 w trybie wyjścia
    gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);

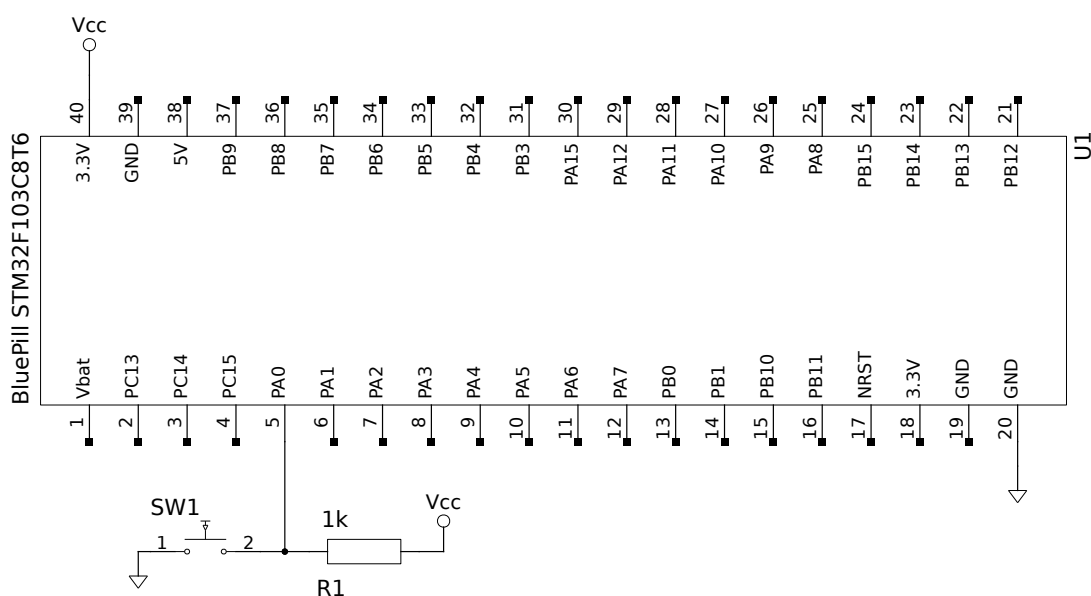
    //Ustawienie pinu A0 w trybie wejścia
    gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_FLOAT, GPIO0);

    int16_t stan_a;

    while(1){
        // odczytaj wartość z portu A
        stan_a = gpio_port_read(GPIOA);
        // Przełącz stan pinu 13 w porcie C bazując na wejściu na porcie A
        if(stan_a & 0x01){
            gpio_set(GPIOC, GPIO13);
        }else{
            gpio_clear(GPIOC, GPIO13);
        }
        // czekamy chwilę ...
        for (int i = 0; i < 150000; i++) __asm__("nop");
    }
}

```

Na płytce stykowej zbuduj układ wg. poniższego schematu:



Po naciśnięciu przycisku na płytce stykowej, powinna zapalić się zielona dioda led. Omówmy teraz powyższy program krok po kroku.

Podobnie jak w poprzednim ćwiczeniu, musimy włączyć używane peryferia włączając ich sygnał zegara. Tym razem używamy dwóch portów GPIO - portu C (do którego jest podłączona dioda) oraz portu A (do którego podłączony jest przycisk). Z tego powodu trzeba uruchomić je oba (kolejność nie ma znaczenia):

```
rcc_periph_clock_enable(RCC_GPIOA);
rcc_periph_clock_enable(RCC_GPIOC);
```

Podobnie jak pin C13, pin A0 też należy skonfigurować<sup>6</sup>. Ustawiamy go jako wejście, bez podciągania (wymuszania jakiegoś stanu, gdy nic nie jest podłączone - mówimy wtedy, że wejście jest pływające):

```
gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);
gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_FLOAT, GPIO0);
```

Same w sobie, wejścia pływające nie mają niczego, co mogło by wymusić na nich stabilny stan logiczny. Zjawiska indukcji elektromagnetycznej (spowodowane m.in. stałym, 50Hz szumem od sieci elektrycznej) indukują na nich losowe napięcia (w pewnym małym zakresie, lecz wystarczająco dużym, aby powodować niestabilne stany logiczne). Z tego powodu nieużywane wejścia z reguły podłącza się do znanego potencjału (GND lub napięcia zasilania układu) aby zapobiegać stanom niestabilnym. W mikrokontrolerach można też włączyć wbudowane rezystory podciągające. Pływającego wejścia można użyć jako źródła liczb losowych<sup>7</sup>

Podobnie jak w poprzednim przykładzie, po zakończeniu inicjalizacji mikrokontrolera (ustawienia GPIO), program wchodzi w nieskończoną pętlę. Za każdą iteracją wczytujemy do niej stan portu A do 16 bitowej zmiennej (w przypadku STM32 port ma 16 bitów szerokości, więc użycie 16-bitowej liczby jest najsensowniejsze):

```
int16_t stan_a;

while(1){
    stan_a = gpio_port_read(GPIOA);
```

W instrukcji warunkowej program sprawdza, czy najmłodszy bit zmiennej jest równy 1. 0x01 to tzw. maska bitowa. Zauważ, że wynik operacji `stan_a & 0x01` jest 1 jeden gdy najmłodszy bit jest równy 1. Tego typu konstrukcje są powszechnie spotykane w programach mikrokontrolerowych.<sup>8</sup> Jeśli warunek wynosi jeden, wykonuje się funkcja `gpio_set`, która gasi diodę. W przeciwnym wypadku, funkcja `gpio_clear` przełącza pin C13 w stan niski, powodując zapalenie diody.

```
if(stan_a & 0x01){
    gpio_set(GPIOC, GPIO13);
}else{
    gpio_clear(GPIOC, GPIO13);
}
```

Następnie wymuszamy opóźnienie (zbyt szybko działający program mógłby mieć problem z drganiem styków):

6. Wejście pływające (*Input*, *Input floating*) jest domyślną konfiguracją wejścia. To wywołanie funkcji nic nie zmienia, ale umieściliśmy je tutaj w ramach przykładu. Jeżeli chcielibyśmy ustawić wewnętrzne podciąganie to zamiast `GPIO_CNF_INPUT_FLOAT` należałoby podać `GPIO_CNF_INPUT_PULL_UPDOWN` i ustawić pin w stanie wysokim lub niskim (w zależności czy chcemy podciągać do stanu wysokiego czy do masy) przy pomocy odpowiednich wywołań funkcji `gpio_set` lub `gpio_clear`.
7. Jest to jedna z prostszych wersji takiego generatora, wcale nie pozbawiona innych wad
8. Jeśli chcielibyśmy sprawdzić inny bit (np. bit na 5 miejscu), użylibyśmy maski, która składa się z samych zer oprócz piątego bitu. Można ją wyrazić jako `0x01 << 5` (jedynka przesunięta o 5 miejsc w lewo). Kompilator zamieni to wyrażenie na stałą.

```
for (int i = 0; i < 150000; i++) __asm__("nop");
```

W ten sposób doszliśmy do końca programu. Pętla jest nieskończona, więc mikroprocesor będzie cyklicznie wykonywał jej zawartość, sterując diodą LED odpowiednio do stanu na A0.

### Zadanie 3.0.1

Zastanów się jakie zmiany należy wykonać w programie 11\_di, aby zamiast dodawać zewnętrzny rezystor podciągający użyć wbudowanego podciągania wejść. Sprawdź swoje przypuszczenia odpowiednio modyfikując układ i program.

### Zadanie 3.0.2

Zastanów się jakie zmiany należy wykonać w programie 11\_di, aby reagował on na przycisk podłączony do pinu A1. Sprawdź swoje przypuszczenia odpowiednio modyfikując układ i program.

### Zadanie 3.0.3

Zmodyfikuj program 11\_di tak, aby przycisk działał jak przełącznik - tzn. jedno naciśnięcie włącza diodę, kolejne ją wyłącza, kolejne znowu włącza, itd.

*Wskazówka: co zawiera zmienna `stan_a` tuż przed załadowaniem jej nowym stanem?*

### Zadanie 3.0.4

Poeksperymentuj z kodem z zadania 3.0.3 modyfikując wartość opóźnienia w pętli głównej. Co się dzieje gdy wartość ta jest zbyt mała? Co jest przyczyną takiego zachowania?

## 4 UART

UART to jeden z popularniejszych protokołów przesyłania danych. Jest podstawą wielu innych protokołów, i obecny praktycznie wszędzie. W tym ćwiczeniu dowiesz się jak zaprogramować peryferium UARTu w swoim mikrokontrolerze. Pokażemy też w jaki sposób “połączyć” znane Ci funkcje wejścia/wyjścia w taki sposób aby odbywało się ono przez UART. Pliki źródłowe znajdują się w katalogu 20\_uart, pierwszy z nich (main.c) ma postać:

```
#include <libopenm3/stm32/rcc.h>
#include <libopenm3/stm32/gpio.h>

#include <stdio.h>
#include "uart.h"

int main(){
    rcc_periph_clock_enable(RCC_GPIOC);
    gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);
    usart_setup();

    while(1){
        for (int i = 0; i < 150000; i++) __asm__("nop");
        gpio_toggle(GPIOC, GPIO13);
        printf("Hello, World!\n");
    }
}
```



Na pierwszy rzut oka ten program wydaje się być bardzo podobny do ćwiczenia pierwszego. Pojawiły się jednak dwa nowe pliki nagłówkowe oraz dwa wywołania nowych funkcji. Pierwszy plik nagłówkowy to znany Ci `stdio.h`, który definiuje funkcje wejścia/wyjścia. Drugi, `uart.h` znajduje się w cudzysłowach, co oznacza że jest to plik lokalny. Poniżej jest jego zawartość:

```
#ifndef _uart_h
#define _uart_h

int _write(int fd, char *ptr, int len);
void usart_setup(void);

#endif
```

W pliku zadeklarowane są dwie funkcje, `_write` oraz `usart_setup`. Przyjrzyjmy się plikowi `uart.c` zawierającemu definicje tych funkcji:

```
#include "uart.h"
#include <libopencm3/stm32/usart.h>
#include <libopencm3/stm32/gpio.h>

void usart_setup(void)
{
    /* Setup GPIO pin GPIO_USART1_TX/GPIO9 on GPIO port A for transmit. */
    gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
                  GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);

    /* Setup UART parameters. */
    usart_set_baudrate(USART1, 9600);
    usart_set_databits(USART1, 8);
    usart_set_stopbits(USART1, USART_STOPBITS_1);
    usart_set_mode(USART1, USART_MODE_TX);
    usart_set_parity(USART1, USART_PARITY_NONE);
    usart_set_flow_control(USART1, USART_FLOWCONTROL_NONE);

    /* Finally enable the USART. */
    usart_enable(USART1);
}

int _write(int fd, char *ptr, int len){
    int i = 0;
    /*
     * Write "len" of char from "ptr" to file id "fd"
     * Return number of char written.
     *
     * Only work for STDOUT, STDIN, and STDERR
     */
    if (fd > 2) {
        return -1;
    }
    while (*ptr && (i < len)) {
        usart_send_blocking(USART1, *ptr);
        if (*ptr == '\n') {
            usart_send_blocking(USART1, '\r');
        }
    }
}
```



```

        }
        i++;
        ptr++;
    }
    return i;
}

```

Funkcja `usart_setup` konfiguruje peryferium USART. Najpierw należy ustawić pin A9 w trybie `GPIO_CNF_OUTPUT_ALTFN_PUSHPULL`, czyli włączyć go jako pin wyjściowy w trybie jego drugiej funkcji (*ALTFN - Alternative Function*). W STM32 większość z pinów pełni dwie funkcje - domyślną z nich jest GPIO, a drugą odpowiednia funkcja peryferium. W celu użycia drugiej funkcji danego pinu, musimy go w taki sposób skonfigurować. W tym przypadku TX znajduje się na pinie A9 (do którego podłączasz programator - ponieważ programowanie STM32 odbywa się przez UART<sup>9</sup>):

```

gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
              GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);

```

Następnie konfigurowany jest sam UART. Tutaj ustawiamy typowe wartości - prędkość 9600 baud<sup>10</sup>, 8 bitów danych, 1 bit stopu. Ustawiamy USART w trybie nadawania (odbieranie jest wyłączone). Ustawiamy brak bitu parzystości i brak kontroli przepływu:

```

/* Setup UART parameters. */
usart_set_baudrate(USART1, 9600);
usart_set_databits(USART1, 8);
usart_set_stopbits(USART1, USART_STOPBITS_1);
usart_set_mode(USART1, USART_MODE_TX);
usart_set_parity(USART1, USART_PARITY_NONE);
usart_set_flow_control(USART1, USART_FLOWCONTROL_NONE);

```

Na końcu procedury uruchamiamy USART:

```

usart_enable(USART1);

```

Przejdźmy teraz do funkcji `_write`. Ta funkcja to nasz STM-owy odpowiednik funkcji `write` z jądra linuxa. Służy ona do wypisania ciągu bajtów o podanej długości na podany deskryptor pliku.

W naszym przypadku ignoruje ona wszystkie deskryptory powyżej 2 (w normalnym środowisku do tych deskryptorów podłączone były by otwarte pliki w systemie plików. Jeśli chcielibyśmy dodać w naszym programie obsługę plików, np. przez kartę SD albo jakiś wirtualny system plików, w tym miejscu należało by wprowadzić pierwszą zmianę):

```

if (fd > 2) {
    return -1;
}

```

9. Można również przez SWD lub JTAG, ale do tego potrzebne są inne programatory. Twoja przejściówka USB-UART pełni tutaj także drugą rolę jako programator

10. *baud* (czyt. bod) to określenie jednostki symbol/sek. Prędkość transmisji w tej jednostce nazywa się *baudrate*. W naszym przypadku symbolem jest stan niski/wysoki (bit). Z reguły *baudrate* nie jest równy przepustowości łącza, ponieważ wlicza się do niego też symbole które nie przenoszą danych (w przypadku UARTa są to bity startu, stopu i ew. parzystości).

Następnie przy pomocy pętli `while`, która wykona się `len` razy lub aż dojdzie do znaku zerowego w napisie<sup>11</sup>, wywołujemy funkcję `usart_send_blocking` z parametrem `USART1` i kolejnymi znakami z wypisywanego napisu. W ten sposób napis podany jako parametr będzie wypisany znak po znaku.

```
while (*ptr && (i < len)) {
    usart_send_blocking(USART1, *ptr);
    if (*ptr == '\n') {
        usart_send_blocking(USART1, '\r');
    }
    i++;
    ptr++;
}
```

Warunek porównujący znak ze znakiem nowej linii i wstawiający znak powrotu karetki jest potrzebny aby po znaku nowej linii kursor w terminalu powrócił na początek linii.

Na końcu zwraca ilość wypisanych znaków (tak jak prawdziwa funkcja `write`):

```
return i;
```

Ta funkcja jest potrzebna (mimo tego, że nigdzie jej bezpośrednio nie wywołujemy) do tego aby funkcje z `stdio.h` mogły komunikować się ze światem. Jak widać jest to ograniczona wersja prawdziwej funkcji `write`, która dostarcza “sztuczną” obsługę plików `stdin`, `stdout`, i `stderr` w naszym środowisku.

To właśnie tą funkcję będą wywoływać `printf`, `puts` w ramach ostatecznego wypisania danych na wyjście.

Wracając do pliku `main.c`:

```
while(1){
    for (int i = 0; i < 150000; i++) __asm__("nop");
    gpio_toggle(GPIOC, GPIO13);
    printf("Hello, World!\n");
}
```

Ta pętla oprócz znanego już migania diodą wypisuje “Hello, World!” na standardowe wyjście (teraz podłączone do UARTa) za każdą iteracją.

Upewnij się, że wszystkie 3 pliki są zapisane a następnie skompiluj i wgraj ten program za pomocą:

```
make
make install
```

a następnie uruchom `picocom` aby zobaczyć co dzieje się na porcie szeregowym:

```
picocom /dev/ttyUSB0
```

#### Porada

Możesz automatycznie uruchamiać `picocom` po udanym wgraniu programu łącząc te dwa polecenia w następujący sposób:

```
make install && picocom /dev/ttyUSB0
```

11. Prawdziwy `write` nie sprawdza tego drugiego warunku, ale ta wersja będzie służyć tylko do wypisywania napisów na UART

## 4.1 Odbieranie danych poprzez UART

W ostatnim ćwiczeniu nauczyliśmy się wysyłać dane poprzez interfejs UART. Uruchomiliśmy także wygodny<sup>12</sup> sposób wypisywania informacji na port szeregowy poprzez standardową funkcję `printf`.

UART możemy wykorzystać także do odbierania danych. W tym ćwiczeniu uruchomimy odbiornik UART, a jako że jego działanie oprzemy na przerwaniach (a nie aktywnym czekaniu na dane w wyznaczonym punkcie programu) to zapoznamy się także z podstawą obsługi przerw.

### Przerwania

Przerwanie jest to (sprzętowy lub programowy) sygnał dla procesora, powodujący zmianę przepływu sterowania, polegającą na przerwaniu aktualnie wykonywanego kodu programu i rozpoczęcie wykonywania procedury obsługi przerwania. Zazwyczaj odbywa się to z użyciem wektora przerw, czyli tablicy używanej do mapowania numeru przerwania na adres pod którym umieszczona jest procedura obsługi danego przerwania.

Na wcześniejszych zajęciach wspomnieliśmy już o przerwaniu zegarowym, które jest używane przez system operacyjny m.in. do okresowego przerywania działania programów celem ustalenia który z czekających procesów powinien rozpocząć swoje (dalsze) wykonywanie. Mechanizm ten (z użyciem przerwania generowanego programowo) jest wykorzystywany także do wywoływania funkcji systemowych z poziomu kodu użytkownika.

Kod źródłowy dla tego ćwiczenia znajduje się w katalogu `21_uart_receiver`. Pliki `uart.h` i `uart.c` mają postać dokładnie taką samą jak poprzednio i nie będziemy ich modyfikować (będą one nam dostarczały jednokierunkowy UART używany przez `printf` w tym i w kolejnych ćwiczeniach). Zmianom uległ natomiast `main.c` i wygląda następująco:

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/stm32/gpio.h>

#include <libopencm3/stm32/usart.h>
#include <libopencm3/cm3/nvic.h>
#include <libopencm3/cm3/scb.h>

#include <stdio.h>

#include "uart.h"
int main() {
    rcc_periph_clock_enable(RCC_GPIOC);
    gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);

    // ustawiamy adres wektora przerw na adres początkowy pamięci FLASH
    SCB_VTOR = FLASH_BASE;
    // aktywujemy przerwania z USART1
    nvic_enable_irq(NVIC_USART1_IRQ);
    // aktywujemy przerwania związane z odbiorem danych z USART1
    usart_enable_rx_interrupt(USART1);

    // aktywujemy USART1 jak wcześniej - parametry transmisji, etc
    usart_setup();
    // aktywujemy pin A10 jako wejściowy
    gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_FLOAT, GPIO10);
```

12. Ta wygoda ma jednak swoją cenę. Jeżeli zauważyliście że ostatni program wgrzywał się dłużej niż poprzednie to jest to efektem zwiększenia jego rozmiaru (a zatem i zapotrzebowania na pamięć flash) na skutek dodania funkcji `printf`. Nasz mikrokontroler posiada jej na tyle dużo iż nie jest to dla nas istotnym problemem, ale warto pamiętać iż użycie `printf` na słabszych mikrokontrolerach może być praktycznie nie do zrealizowania.

```

// zmieniamy tryb na TX/RX
usart_set_mode(USART1, USART_MODE_TX_RX);

for(int j = 0; j < 10; j++) {
    printf("...%d\n", j);
    for (int i = 0; i < 150000; i++) __asm__("nop");
    gpio_toggle(GPIOC, GPIO13);
}

while(1) {
    __asm__("nop");
}

void usart1_isr(void) {
    uint32_t flags = USART_SR(USART1);
    if ( flags & USART_SR_RXNE ) {
        // przerwanie było z powodu odebranego bajtu
        uint8_t data = usart_recv(USART1);
        if (data%2)
            gpio_set(GPIOC, GPIO13);
        else
            gpio_clear(GPIOC, GPIO13);
    }
}

```

Pierwszą rzeczą konieczną do wykonania jest ustawienie adresu wektora przerwań na początkowy adres pamięci flash, gdyż nasz program wgrywany i uruchamiany jest właśnie z wbudowanej pamięci flash, a kompilator wektor przerwań umieszcza na początku kodu<sup>13</sup>:

```
SCB_VTOR = FLASH_BASE;
```

Następnie możemy i powinniśmy włączyć obsługę (odbieranie) przerwań związanych z USART1:

```
nvic_enable_irq(NVIC_USART1_IRQ);
```

oraz aktywować generowanie przez układ interfejsu USART przerwań związanych z odebraniem bajtu:

```
usart_enable_rx_interrupt(USART1);
```

W kolejnych krokach konfigurujemy UART tak jak w poprzednim ćwiczeniu, konfigurujemy pin A10 (USART1 RX) jako input oraz przełączamy tryb pracy naszego portu szeregowego na nadawanie i odbiór:

```
usart_setup();
gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_FLOAT, GPIO10);
usart_set_mode(USART1, USART_MODE_TX_RX);
```

Po powitalnym odliczaniu funkcje main kończymy nieskończoną, pustą (wykonującą nop) pętlą while:

13. W tablicy tej, na dobrze zdefiniowanej pozycji (bajty 4-7, odpowiadające wyjątkowi *reset*), umieszczany jest także adres początku kodu związanego z naszym programem. To do tego adresu wykonywany jest skok gdy zrestartujemy nasz mikrokontroler z ustawioną zworkami BOOT opcją uruchamiania z pamięci flash lub wydamy wbudowanemu bootloaderowi polecenie uruchomienia od początku pamięci flash (opcja -g 0x0).

```
while(1)
    _asm_>("nop");
```

Musimy dodać jeszcze jedną funkcję która będzie związana z obsługą przerwania od portu szeregowego. Używana biblioteka wymaga aby funkcja ta nazywała się `usart1_isr`. W ramach niej odczytujemy z bufora wejściowego odebrany bajt (przy użyciu funkcji `usart_recv()`) i w zależności od jego parzystości włączamy lub wyłączamy LED podłączony do pinu C13:

```
void usart1_isr(void) {
    if ( USART_SR(USART1) & USART_SR_RXNE ) {
        uint8_t data = usart_recv(USART1);
        if (data%2)
            gpio_set(GPIOC, GPIO13);
        else
            gpio_clear(GPIOC, GPIO13);
    }
}
```

#### Zadanie 4.1.1

Zmodyfikuj program `21_uart_receiver` tak aby odbierał poprzez port szeregowy liczbę w zakresie od 1 do 9 i odpowiadał na nią trójkątem z gwiazdek odpowiedniej wielkości. Na przykład gdy otrzyma "3" powinien to być:

```
*
**
***
```

#### Zadanie 4.1.2

Zmodyfikuj rozwiązanie zadania 4.1.1 tak aby obsługiwać liczby wielocyfrowe (możemy ograniczyć się np. do dwucyfrowych). Odczyt liczby powinien kończyć w momencie wczytania znaku nowej linii. Wtedy też powinno nastąpić generowanie trójkąta z gwiazdek.

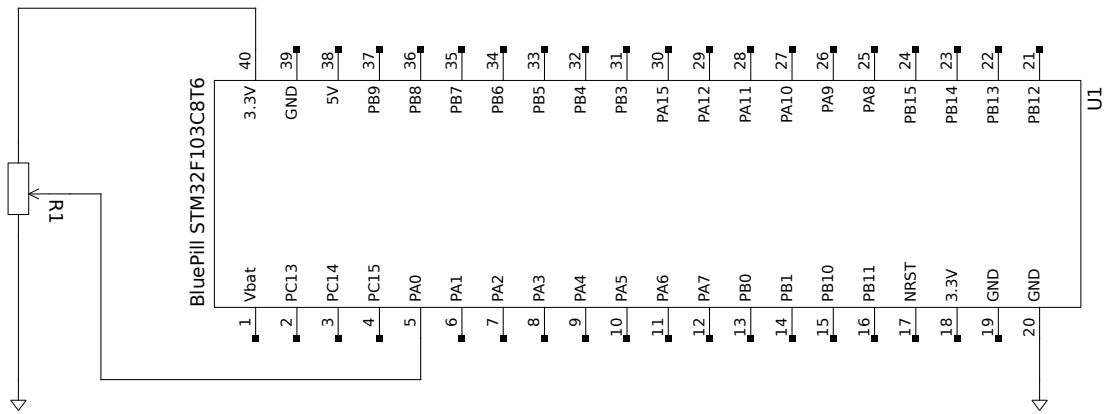
*Wskazówka 1: wartość pierwszej cyfry przechowuj w zmiennej globalnej do czasu otrzymania kolejnej cyfry*

*Wskazówka 2: zwróć uwagę na kodowanie nowej linii - '\n' vs '\r\n' vs '\n\r', etc*

## 5 Przetwornik analogowo-cyfrowy

Przetwornik analogowo-cyfrowy w STM32 należy do jednych z bardziej zaawansowanych. W tym ćwiczeniu pokażemy prosty przykład użycia przetwornika ADC do odczytu napięcia z jednego z pinów mikrokontrolera.

Zbuduj układ zgodnie z poniższym schematem:



Program będzie wysyłał wynik konwersji na UART. Do wysyłania komunikatów na uart użyjemy kodu z poprzedniego ćwiczenia. Kod źródłowy znajduje się w katalogu 30\_adc i wszystkie elementy związane z obsługą ADC umieszczone zostały w pliku main.c.

Najpierw musimy zainicjalizować peryferium ADC. W tym celu stworzyliśmy funkcję `adc_setup()`:

```
void adc_setup(){
    //Ustawienie A0 w tryb wejścia analogowego
    gpio_set_mode(GPIOA, GPIO_MODE_INPUT, GPIO_CNF_INPUT_ANALOG, GPIO0);

    //Wyłączenie ADC
    adc_power_off(ADC1);

    //Konfiguracja dla trybu pojedynczych pomiarów
    adc_disable_scan_mode(ADC1);
    adc_set_single_conversion_mode(ADC1);
    adc_disable_external_trigger_regular(ADC1);
    adc_set_right_aligned(ADC1);
    adc_set_sample_time_on_all_channels(ADC1, ADC_SMPR_SMP_28DOT5CYC);

    //Uruchomienie ADC i chwila przerwy na start
    adc_power_on(ADC1);
    for (int i = 0; i < 150000; i++) __asm__("nop");

    //Rekalibracja ADC
    adc_reset_calibration(ADC1);
    adc_calibrate(ADC1);
}
```

Powyższy fragment kodu ustawia pin A0 w tryb wejścia analogowego i konfiguruje ADC w sposób pozwalający na przeprowadzanie pojedynczych konwersji. Następnie włącza ADC, czeka pewien okres czasu aż ADC się ustabilizuje wykonuje procedurę kalibracji.

Teraz możemy zdefiniować funkcję, która zwróci nam wynik pojedynczej konwersji.

```
uint16_t adc_read(){
    //Ustawiamy numery kanałów na których ma być przeprowadzona konwersja
    uint8_t channels[16];
    channels[0] = 0;
    adc_set_regular_sequence(ADC1, 1, channels);
}
```

```

//Uruchomienie konwersji
adc_start_conversion_direct(ADC1);
//Oczekiwanie na koniec konwersji
while(!adc_eoc(ADC1));
//Zwrócenie wartości
return adc_read_regular(ADC1);
}

```

Peryferium ADC w mikrokontrolerach STM32 pobiera listę kanałów na których ma się odbyć konwersja. W naszym przypadku chcemy pobrać wartość tylko z jednego kanału. Pin A0 to kanał zerowy, więc umieszczemy zero w pierwszym elemencie tablicy którą prześlemy funkcji ustawiającej kolejność konwersji. Do tej funkcji przekazujemy również ilość kanałów dla których przeprowadzana będzie konwersja - w tym przypadku 1.

Następnie uruchamiamy konwersję i oczekujemy na jej zakończenie (`adc_eoc()` zwraca fałsz dopóki trwa konwersja). Na końcu naszej funkcji zwracamy wynik konwersji (wynik funkcji `adc_read_regular(ADC1)`).

Mając już gotowe funkcje obsługujące ADC pozostaje nam tylko włączyć jego zegar i użyć w jakiś sposób wyniku naszej konwersji:

```

int main(){
    rcc_periph_clock_enable(RCC_GPIOC);
    rcc_periph_clock_enable(RCC_ADC1);

    gpio_set_mode(GPIOC, GPIO_MODE_OUTPUT_2_MHZ, GPIO_CNF_OUTPUT_PUSHPULL, GPIO13);
    usart_setup();
    adc_setup();
    while(1){
        for (int i = 0; i < 150000; i++) __asm__("nop");
        gpio_toggle(GPIOC, GPIO13);
        printf("ADC: %d\n", adc_read());
    }
}

```

W `main` pojawiło się wywołanie `rcc_periph_clock_enable` uruchamiające zegar dla ADC, wywołanie naszej funkcji konfiguracyjnej ADC oraz wypisanie wyniku konwersji w pętli za pomocą `printf()`.

Po uruchomieniu programu powinien zacząć migać zielony LED, a po uruchomieniu `picocom` powinniśmy widzieć wyniki konwersji przesyłane przez UART. Obracając potencjometr zmieniamy napięcie na pinie A0, co zmienia wynik konwersji ADC.

#### Zadanie 5.0.1

Wiedząc, że wartość 4096 odpowiada napięciu 3.3V, a 0 napięciu 0V, zmień przykładowy program ADC (`30_adc`) tak, aby zamiast surowej wartości wypisywał wartość napięcia.

*Wskazówka: Aby uprościć obliczenia (uniknąć działań na liczbach zmiennoprzecinkowych), Twój program może podawać wartość w mV.*

#### Zadanie 5.0.2

Zmodyfikuj rozwiązanie zadania 5.0.1 tak aby wypisywało wynik w V, bez stosowania arytmetyki zmiennoprzecinkowej.

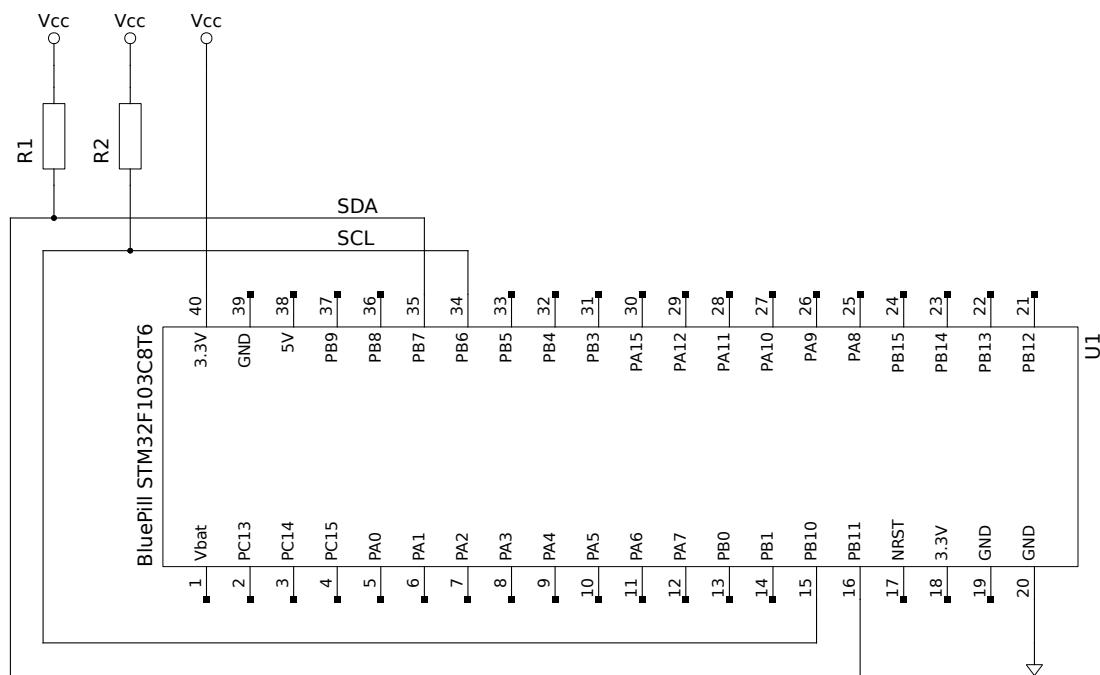
*Wskazówka: Wypisz osobno część całkowitą i część ułamkową*



## 6 Komunikacja $I^2C$

$I^2C$  jest popularnym protokołem szeregowym wykorzystywanym do komunikacji między cyfrowymi układami scalonymi. Jego odmiana, SMBus, jest m. in. stosowana do komunikacji z inteligentnymi bateriami. W tym ćwiczeniu zademonstrujemy implementację mastera oraz slave'a  $I^2C$  na STM32.

W modelu STM32 używanym przez nas są dwa peryferia  $I^2C$ . Połączymy je ze sobą i oprogramujemy jedno jako master, a drugie jako slave. Dzięki temu nasz mikrokontroler będzie mógł „rozmawiać” sam ze sobą<sup>14</sup> poprzez magistralę  $I^2C$ . Nasz układ wygląda tak:



Rezystory podciągające powinny mieć wartość kilku  $k\Omega$ .

Kod do tego ćwiczenia znajduje się w katalogu 40\_i2c. Pliki `uart.*` nie uległy zmianie od kiedy zaczęliśmy ich używać i dostarczają obsługę wypisywania informacji na port szeregowy. Całość obsługi  $I^2C$  znajduje się w pliku `main.c` - przyjrzyjmy mu się bliżej.

Zaczynamy od zdefiniowania adresu naszego slave'a - zdefiniujemy makro:

```
#define SLAVE_ADDR 0x0F
```

Następnie należy skonfigurować oba peryferia. Funkcja inicjalizująca może zostać zaimplementowana w taki sposób:

```
void i2c_setup(){
    rcc_periph_clock_enable(RCC_GPIOB);
    rcc_periph_clock_enable(RCC_I2C1);
    rcc_periph_clock_enable(RCC_I2C2);

    /* I2C1 - master; SDA=B7, SCL=B6 */
    i2c_reset(I2C1);
    i2c_peripheral_disable(I2C1);
    i2c_set_speed(I2C1, i2c_speed_sm_100k, 8);
}
```

14. Oczywiście w realnych zastosowaniach nie stosuje się takiego połączenia, jednak jest to dobry przykład do pokazania obsługi  $I^2C$  zarówno od strony mastera jak i slave'a, bez konieczności używania jakichkolwiek innych układów.

```

gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_2_MHZ,
              GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN, GPIO6 | GPIO7);

i2c_peripheral_enable(I2C1);

/* I2C2 - slave; SDA=B11, SCL=B10 */
i2c_reset(I2C2);
i2c_peripheral_disable(I2C2);
i2c_set_speed(I2C2, i2c_speed_sm_100k, 8);

i2c_set_own_7bit_slave_address(I2C2, SLAVE_ADDR);

nvic_enable_irq(NVIC_I2C2_EV_IRQ);

gpio_set_mode(GPIOB, GPIO_MODE_OUTPUT_2_MHZ,
              GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN, GPIO10 | GPIO11);

i2c_enable_interrupt(I2C2, I2C_CR2_ITVTEN );
i2c_peripheral_enable(I2C2);

i2c_enable_ack(I2C2);
}

```

Powyższa funkcja konfiguruje oba interfejsy. Konfigurację zaczynamy włączenia taktowania, resetu oraz wyłączenia peryferiów (konfiguracje wstępne powinny odbywać się przy wyłączonych peryferiach). Pierwszy interfejs (I2C1) będzie masterem, zaś drugi (I2C2) - slave'm.

W przypadku mastera ustawiamy tylko prędkość magistrali za pomocą funkcji `i2c_set_speed(I2C1, i2c_speed_sm_100k, 8)`, która jako argumenty pobiera peryferium (I2C1), prędkość (jako wartość typu wyliczeniowego enum), oraz prędkość zegara taktującego peryferium w MHz (domyślnie jest to *HSI Clock*, wewnętrzny oscylator - w naszym przypadku 8MHz).

Dla I2C2 konfiguracja wymaga także ustawienia adresu (funkcja `i2c_set_own_7bit_slave_address(I2C2, SLAVE_ADDR)`), włączenia przerwania `nvic_enable_irq(NVIC_I2C2_EV_IRQ)` i skonfigurowania go (`i2c_enable_interrupt(I2C2, I2C_CR2_ITVTEN )`), generowanie przerwania dla wszystkich wydarzeń na szynie danych) oraz włączenia odpowiedzi na własny adres (odsyłanie *ACK*, funkcja `i2c_enable_ack(I2C2)`, wywoływana po włączeniu peryferium).

Oprócz wymienionych kroków należy też skonfigurować piny GPIO na których (domyślnie) znajdują się sygnały  $I^2C$ , w trybie `GPIO_CNF_OUTPUT_ALTFN_OPENDRAIN` (*open-drain*, alternatywna funkcja).

Ponieważ nasz procesor jest jednowątkowy, obsługa slave'a będzie odbywać się całkowicie w funkcji obsługi przerwania. W związku z tym musimy ją zdefiniować. Nasza biblioteka oraz skrypty linkera oczekują, że będzie nazywać się `i2c2_ev_isr`.

```

void i2c2_ev_isr(void){
    uint16_t sr1, sr2;

    sr1 = I2C_SR1(I2C2);
    printf("I2C ISR: sr1=0x%04x slavebyte=%d\n", sr1, slavebyte); // for demonstration
    ↪ / debug only
}

```

```

// Address matched (Slave)
if (sr1 & I2C_SR1_ADDR){
    // Clear the ADDR sequence by reading SR2.
    sr2 = I2C_SR2(I2C2);
    (void) sr2;
}

// Write request from master
if (sr1 & I2C_SR1_RxNE){
    slavebyte = I2C_DR(I2C2);
    printf("I2C ISR: received %d\n", slavebyte); // for demonstration / debug only
    slavebyte *= 2; // slave logic is multiply by 2 ;)
}

// Read request from master
if ((sr1 & I2C_SR1_TxE)){
    I2C_DR(I2C2) = slavebyte;
}

// Stop sequence detected (Slave)
if (sr1 & I2C_SR1_STOPF){
    // Clear by write I2C_CR1
    I2C_CR1(I2C2) = I2C_CR1(I2C2);
}
}

```

Ta procedura obsługi przerwania wykonuje się dla wszystkich wydarzeń na szynie danych. W związku z tym należy najpierw sprawdzić, jakie wydarzenie spowodowało wystąpienie przerwania. Służy do tego rejestr I2C\_SR1 (*Status Register 1*). Jego adres zwraca makro I2C\_SR1(x), które oblicza go na podstawie adresu bazowego peryferium. W zależności od tego, które wydarzenie na szynie danych wygenerowało przerwanie, ustawiane są odpowiednie bity w tym rejestrze.

Jeżeli przerwanie zostało wygenerowane poprzez odebranie własnego adresu, spełniony będzie warunek `sr1 & I2C_SR1_ADDR` (w rejestrze ustawiony będzie bit I2C\_SR1\_ADDR). W takim wypadku peryferium oczekuje od programu odczytania rejestru SR2.

#### Uwaga

Odczytanie rejestru SR2 powoduje wysłanie ACK i przejście interfejsu do stanu gotowości do odebrania danych - inaczej peryferium będzie trzymać linię SCL nisko, blokując szynę. Tego typu model programowy, w którym odczytanie lub zapisanie wartości rejestru powoduje jakąś akcję peryferium jest często spotykany. Narzuca on pewne ograniczenia ale ma też swoje zalety, m. in. pozwala na szybszą obsługę takich interfejsów. W przypadku interfejsu  $I^2C$  trzeba odczytać i/lub zapisać pewne rejestry po większości operacji na szynie.

Następnie procedura sprawdza, czy rejestr danych (DR) nie jest pusty (ustawiony bit I2C\_SR1\_RxNE w SR1, *Rx Not Empty*). Jeśli tak jest, to znaczy że odebrano bajt z szyny danych (od mastera). Należy wtedy wczytać ten bajt z DR

Tutaj także wykonujemy operację którą realizuje slave. W tym przypadku jest nią pomnożenie odebranego bajtu razy 2 i zachowanie go w pamięci.

Jeśli master wyśle żądanie odczytu ze slave'a, po otrzymaniu adresu, w SR1 ustawiony będzie bit I2C\_SR1\_TxE (*Tx Empty*) informujący o tym, że takie żądanie nastąpiło a rejestr danych do wysłania jest pusty. Wtedy zapisanie bajtu danych do DR spowoduje wysłanie tych danych na szynę do mastera. W naszym przypadku daną do wysłania jest poprzednio zachowana i pomnożona wartość.

Jeżeli master wykona na szynie sekwencję stopu, co jest to sygnalizowane ustawieniem bitu I2C\_SR1\_STOPF w SR1, to należy potwierdzić jej odebranie poprzez dokonanie zapisu do rejestru CR1. Wykonuje to linijka I2C\_CR1(I2C2) = I2C\_CR1(I2C2). Mimo braku jej sensu z punktu widzenia programu, należy pamiętać, że makra rejestrów definiują je jako volatile, więc nie zostanie ona usunięta w procesie optymalizacji kodu, a kompilator wygeneruje kod zapisujący wartość do CR1.<sup>15</sup>

#### Uwaga

Procedura dodatkowo wypisuje na porcie szeregowym informację o tym że została wywołana i jakie dane odebrała. Ogólnie nie należy stosować tak powolnych operacji jak nasza funkcja printf w procedurach obsługi przerw, gdyż może to prowadzić do ich niewłaściwego działania. Wypisywanie zostało tutaj wprowadzone wyłącznie w celach demonstracyjnych i może (a nawet powinno) być usunięte przed jakimkolwiek poważniejszym użyciem tego kodu.

Oprogramowanie interfejsu mastera jest mniej skomplikowane, ponieważ odbywa się w zwykłej funkcji. Poniżej funkcja zapisu bajtu do slave'a (jako argument pobiera interfejs oraz wartość bajtu):

```
void i2c_send_write(uint32_t peryf, uint8_t dane){
    i2c_send_start(peryf);
    // Czekaj na wystanie startu
    while (!(I2C_SR1(peryf) & I2C_SR1_SB
            & (I2C_SR2(peryf) & (I2C_SR2_MSL | I2C_SR2_BUSY))));

    i2c_send_7bit_address(peryf, SLAVE_ADDR, I2C_WRITE);
    //Czekaj na wystanie adresu
    while (!(I2C_SR1(peryf) & I2C_SR1_ADDR));
    (void) I2C_SR2(peryf); //Wyczyść EV6

    i2c_send_data(peryf, dane);

    while (!(I2C_SR1(peryf) & (I2C_SR1_BTF))); //Czekaj na wysłanie danych

    i2c_send_stop(peryf);
}
```

Transmisja danych rozpoczyna START, którego wysłanie powoduje funkcja i2c\_send\_start(peryf). Następnie oczekujemy na zakończenie wysyłania START wykonując pustą pętlę dopóki bit I2C\_SR1\_SB (*Start Bit*) w SR1 oraz bity I2C\_SR1\_MSL (*Master/slave*) i I2C\_SR1\_BUSY (oznaczający, że na szynie odbywa się komunikacja) w SR2 nie zostaną ustawione na 1. Ponieważ funkcja i2c\_send\_start(peryf) (oraz inne funkcje związane z kontrolą I2C) ustawia tylko bity rejestrach, zakończy się ona wcześniej niż na szynie pojawi się jej oczekiwany efekt. Z tego powodu zawsze należy poczekać, aż żądane działanie rzeczywiście się zakończy.

Po wysłaniu warunku START, wysyłamy adres z flagą R/W ustawioną na zapis, za pomocą funkcji i2c\_send\_7bit\_address(peryf, SLAVE\_ADDR, I2C\_WRITE). Po tej operacji należy poczekać aż bit I2C\_SR1\_ADDR w SR1 zostanie ustawiony, a następnie wykonać odczyt SR2<sup>16</sup>(linijka (void) I2C\_SR2(peryf) odczytuje tę wartość a następnie ją porzuca. Jest to obowiązkowe). Potem wysyłamy bajt za pomocą funkcji i2c\_send\_data(peryf, dane) i oczekujemy na ustawienie bitu I2C\_SR1\_BTF (*Byte Transfer Finished* w SR1. Operację zapisu kończymy wysyłając STOP za pomocą i2c\_send\_stop(peryf) (tutaj wyjątkowo nie trzeba czekać na ustawienie żadnych flag).

15. Jeżeli tego nie zrobimy procesor będzie ciągle generować przerwanie obsługi I2C z ustawionym bitem I2C\_SR1\_STOPF w SR1 i nasz program zawiesi się.

Podobnie możemy stworzyć funkcję odczytu bajtu ze slave'a:

```
uint8_t i2c_send_read(uint32_t peryf){
    uint8_t dane;

    i2c_send_start(peryf);
    // Czekaj na wystanie startu
    while (!(I2C_SR1(peryf) & I2C_SR1_SB
            & (I2C_SR2(peryf) & (I2C_SR2_MSL | I2C_SR2_BUSY))));

    i2c_send_7bit_address(peryf, SLAVE_ADDR, I2C_READ);

    //Czekaj na wystanie adresu
    while (!(I2C_SR1(peryf) & I2C_SR1_ADDR));
    (void) I2C_SR2(peryf); //Wyczyść EV6

    //Czekaj aż otrzymasz 1 bajt danych
    while (!(I2C_SR1(peryf) & I2C_SR1_RxNE));
    dane = i2c_get_data(peryf);

    i2c_send_stop(peryf);

    return dane;
}
```

Z taka różnicą, że adres wysyłamy z flagą odczytu (I2C\_READ), a po wysłaniu adresu czekamy na zapelnienie się rejestru danych bajtem od slave'a (ustawienie bitu I2C\_SR1\_RxNE w SR1). Funkcja i2c\_get\_data(peryf) odczytuje rejestr DR. Operację kończymy wysyłając STOP.

Mając zdefiniowane funkcje - konfigurującą interfejsy, wysyłającą i odbierającą dane, oraz realizującą logikę slave'a (funkcja obsługi przerwania I2C2) - możemy ich użyć do napisania programu testowego. Skorzystamy też z poprzednich funkcji do komunikacji po UART aby wyświetlić wyniki naszego programu:

```
uint8_t k = 0;
i2c_setup();

while(1){
    for (int i = 0; i < 500000; i++) __asm__("nop");

    printf("Wysylam %d\n", k);
    i2c_send_write(I2C1, k);
    uint8_t wynik = i2c_send_read(I2C1);
    printf("Odebralem %d\n", wynik);
    k++;
}
```

Ten kod wysyła kolejne liczby całkowite do slave'a i wypisuje jego odpowiedź na port szeregowy mikronotrolera.

---

16. Dokładna kolejność działań na rejestrach dla periferium I<sup>2</sup>C jest opisana w rozdziałach 26.3.2 (oprogramowanie slave'a) i 26.3.3 (oprogramowanie mastera) w *Reference manual*

Należy pamiętać, że wyjścia I2C są typu otwarty kolektor i potrzebują rezystorów podciągających!

### Zadanie 6.0.1

Zmień funkcję realizującą logikę slave'a w przykładowym kodzie I2C (40\_i2c) tak, aby zamiast mnożyć otrzymaną liczbę przez 2, dodawał do niej jakąś (dowolną - ustaloną przez programistę) stałą.

### Zadanie 6.0.2

Zapoznaj się z dokumentacją posiadanego rejestru z interfejsem I2C. Podłącz swój układ do magistrali I2C1 mikrokontrolera (pin B7 jako SDA, pin B6 jako SCL). Zmodyfikuj program 40\_i2c tak aby obsługiwać posiadany rejestr - jako układ wejścia (odczyt stanu jego GPIO) oraz wyjścia (ustawianie stanu jego GPIO).

*Wskazówka: STM32 będzie działać jako master magistrali I2C, a slawem będzie układ IO. Zatem nie będziesz potrzebować części kodu związanego z obsługą slave, działającego na I2C2.*

## 7 Komunikacja z Linuxem ☺

### 7.1 I<sup>2</sup>C w komputerach

Magistrala ta jest powszechnie używana w komputerach stacjonarnych i laptopach, jednak trudno w nich dostać się do odpowiednich połączeń. Z magistrali tej bez problemów możemy korzystać także w komputerach jednopłytkowych typu Pi (gdzie znajduje się na złączu GPIO) obsługiwanych z poziomu Linuxa.

Linux dostarcza narzędzi do obsługi magistrali I<sup>2</sup>C - zarówno z linii poleceń jak i z poziomu kodu źródłowego własnego programu.

#### 7.1.1 narzędzia linii poleceń

Narzędzia linii poleceń dostarczane są przez pakiet *i2c-tools*. Podstawowe dwa polecenia służą do odczytu i zapisu wskazanego rejestru urządzenia I2C:

- `i2cget szynai2c adres_ukladu adres_rejestru` – odczytuje rejestr o adresie `adres_rejestru`, z układu I2C o adresie `adres_ukladu` znajdującego się na wskazanej magistrali I2C (`szynai2c`)
- `i2cset szynai2c adres_ukladu adres_rejestru wartosc` – zapisuje podaną wartość do określonego rejestru układu I2C

Pakiet dostarcza także inne polecenia, z których należy wspomnieć o:

- `i2cdetect -l` – listuje magistrale I2C
- `i2cdetect szynai2c` – listuje urządzenia na wskazanej magistrali I2C
- `i2cdump szynai2c adres_ukladu` – listuje rejestry wskazanego układu I2C

**Uwaga:** Użycie tych poleceń może być niebezpieczne, zwłaszcza gdy nie wiemy jakie układy znajdują się na danej magistrali I2C. Związane jest to z tym iż odczyt rejestru o danym numerze wiąże się z operacją zapisu do układu I2C, a taka (przez układ nie obsługujący adresowania rejestrowego) może być zinterpretowana jako zwykły zapis do układu. Dlatego odradzamy eksperymentowanie z tymi poleceniami w laptopach i komputerach stacjonarnych.

#### 7.1.2 kod C

Oczywiście możliwa jest też obsługa tej magistrali z poziomu własnego kodu języka C lub C++ przy użyciu odpowiednich wywołań funkcji bibliotecznych. Przykładowy kod C/C++ demonstrujący użycie komunikacji I2C znajduje się na [http://vip.opcode.eu.org/#komunikacja\\_I2C](http://vip.opcode.eu.org/#komunikacja_I2C).



Typowo instalacja Linuxa dla komputerka jedнопłytkowego typu \*Pi lub podobnego sprowadza się do nagrania pobranego obrazu systemu (Raspbian dla Raspberry Pi, Armbian dla prawie całej reszty) na kartę SD z użyciem np. polecenia `dd`. Możliwe są także inne metody przygotowania obrazu, np. z użyciem `qemu-debootstrap` pozwalające na dostosowanie tworzonych obrazu<sup>a</sup>.

Główną różnicą w stosunku do systemu na komputerach stacjonarnych i laptopach jest proces startu – nie mamy tutaj doczynienia z BIOSem i GRUBem. Zamiast tego mikrokontroler znajdujący się na płycie posiada wbudowany jakiś bootloader, który typowo potrafi załadować kolejny bootloader z karty SD i przekazać do niego sterowanie.

W przypadku Raspberry ten pierwszy etap bootloadera na karcie SD znajduje się w pliku (identyfikowanym nazwą na partycji typu FAT) `bootcode.bin`, a kolejny to `start.elf` (korzystający z plików `config.txt`, `cmdline.txt` i `kernel.img`).

W przypadku większości innych płytek wykorzystywany jest bootloader *U-Boot* wgrywany na kartę SD (poza systemem plików na niej umieszczonym). Rozwiązanie trochę przypominające to z czym mamy do czynienia w komputerach typu PC. Bootowanie jednak na ogół nie rozpoczyna się od początku karty SD (MBR), a od określonego miejsca na tej karcie (np. 8kiB dla sunxi), gdzie umieszczany jest *U-Boot SPL*. Zajmuje się on m.in. inicjalizacją pamięci i przekazuje sterowanie do właściwego kodu *U-Boot* znajdującego się w dalszej części karty SD (także poza jakąkolwiek partycją – typowo na 40kiB karty SD, pomiędzy MBR a pierwszą partycją). Dopiero ten kod ładuje pliki konfiguracyjne i obrazy jądra z odpowiedniej partycji na karcie SD.<sup>b</sup>

Po załadowaniu jądra wykorzystywany jest typowy `systemd`<sup>c</sup> znany z „Linuxa” na komputerach PC, który możemy zastąpić nawet zwykłym skryptem `sh`. Po uruchomieniu otrzymujemy typowy system typu Debian GNU/Linux, konfigurowany przy pomocy standardowych wpisów w `/etc`, `/proc` oraz typowych narzędzi (jak np. `systemctl`, `ip`, `apt`). Różnicą jest dostęp do pinów GPIO mikrokontrolera oraz peryferiów na nich dostępnych.

Za mapowanie poszczególnych peryferiów na piny GPIO (czyli za to czy na danym pinie mamy np. UART-TX, I2C-SDA czy zwykłe GPIO) odpowiada w Linuxie mechanizm *Device Tree*. Systemy takie jak Raspbian czy Armbian dostarczają zestawy plików DTB aktywujących odpowiednie mapowania i celem uzyskania odpowiedniego peryferium należy je aktywować w konfiguracji startowej (`/boot/config.txt` w Raspbianie, `/boot/armbianEnv.txt` w Armbianie). Na przykład:

```
dtparam=i2c_arm=on
dtoverlay=i2c-bcm2708
```

dopisany do `/boot/config.txt` aktywuje interfejs I2C na starszych płytkach *Raspberry Pi*. Natomiast dodanie `i2c0` do linijki `overlays=` w `/boot/armbianEnv.txt` aktywuje I2C na pinach 3 i 5 (PA12, PA11) na *Orange Pi Zero*. Do aktywacji odpowiednich DTB w Armbianie można użyć polecenia `armbian_config`, następnie wejść w System i Hardware, gdzie można aktywować / dezaktywować takie mapowania. Więcej informacji na temat *Device Tree overlays*: [https://docs.armbian.com/User-Guide\\_Allwinner\\_overlay\\_s/](https://docs.armbian.com/User-Guide_Allwinner_overlay_s/).

Kombinacje niektórych modeli Orange Pi z niektórymi switchami mają problemy z mechanizmem autonegocjacji prędkości łącza ethernetowego. Rozwiązaniem jest wymuszenie prędkości 100Mbit/s po obu stronach lub (jeżeli nie mamy możliwości konfigurowania switcha) ustawienie na OrangePi prędkości 10Mbit/s poleceniem: `ethtool -s eth0 speed 10 duplex full autoneg off`.

- 
- a. Taką metodą tworzenia lekkiego obrazu dla Raspberry Pi opisałem trochę szerzej na <http://blog.opcode.eu.org/2020/05/15/rpi-debootstrap.html>.
  - b. Więcej na temat bootowania można dowiedzieć się np. z [https://linux-sunxi.org/Bootable\\_SD\\_card](https://linux-sunxi.org/Bootable_SD_card).
  - c. Więcej informacji na <http://www.opcode.eu.org/SystemBoot.xhtml#systemd>



## 7.2 UART

### 7.2.1 I<sup>2</sup>C vs UART

I<sup>2</sup>C jest bardzo popularną magistralą "lokalną" używaną do połączenia CPU z układami peryferyjnymi (którymi mogą być np. także inne mikrokontrolery). Ma ona ograniczony zasięg – typowo kilkanaście / kilkadziesiąt cm, raczej nie spotyka się kilkumetrowych rozwiązań. Na większych odległościach tego typu funkcje pełni UART<sup>17</sup>, najczęściej w wariantcie RS485, pozwalającego na łączenie do 32 urządzeń na jednej linii. W odróżnieniu od I2C sam UART nie określa żadnego sposobu adresacji i wykorzystywane są do tego inne protokoły, których powyżej UART/RS485 jest naprawdę bardzo dużo. Chyba najpopularniejszym jest Modbus RTU.

### 7.2.2 Modbus – adresacja i identyfikacja ramki

Modbus jest otwartym, prostym protokołem komunikacyjnym występującym w kilku odmianach – RTU, ASCII, TCP. Dwie pierwsze wykorzystują łącze szeregowe typu UART, trzecia używa pakietów IP/TCP. Protokół Modbus:

1. jest protokołem master-slave, czyli jest wyraźnie określone, które urządzenie inicjalizuje transmisję, a które jedynie odpowiada na otrzymane żądania
2. zapewnia adresację urządzeń (8 bitowy adres, zakres 1–247)
3. określa sposób dostępu do danych w urządzeniu (określany 8 bitowym kodem funkcji) i adresację tych danych (16 bitowy adres rejestru / wejścia binarnego / ...)
4. określa sposób identyfikacji początku i końca ramki

Ten ostatni punkt jest szczególnie istotny przy przesyłaniu danych z użyciem łącza typu UART, gdzie identyfikowany jest tylko początek/koniec bajtu, ale nie ma określonego sposobu identyfikacji początku/końca grupy bajtów którą jest np. ramka jakiegoś protokołu.

Modbus ASCII liczby reprezentujące adresy, kody funkcji i dane zapisuje w postaci tekstowej (jako liczby szesnastkowe), więc do identyfikacji początku i końca ramki mogą posłużyć inne niż (0-9A-F) znaki ASCII – początek ramki oznacza dwukropek (:), a koniec ciąg `\r\n`.

Modbus RTU<sup>18</sup> przesyła te wartości liczbowe binarnie (czyli po prostu w postaci danej liczby), zatem nie ma tutaj żadnej wolnej wartości, którą można by użyć jako znacznik początku / końca ramki. W tym przypadku służą do tego zależności czasowe:

- odstęp pomiędzy bajtami w ramce nie może przekroczyć 1.5 czasu trwania transmisji jednego bajtu
- odstęp pomiędzy ramkami musi wynosić przynajmniej 3.5 czasu trwania transmisji jednego bajtu

Pełną specyfikację protokołu można znaleźć na <https://modbus.org/specs.php>.

### 7.2.3 Modbus w Linuxie

Obsługę protokołu Modbus RTU po stronie Linuxa może zapewnić program `mbpoll` pozwalający zarówno na odczyt (modbusowe funkcje 1, 2, 3 i 4) jak i zapis (modbusowe funkcje 5 i 6) danych z/do urządzenia modbus TCP lub RTU.

Jeżeli ten zestaw funkcjonalności jest dla nas niewystarczający lub po prostu potrzebujemy zintegrować obsługę *Modbus* z naszym kodem programu możemy użyć bezpośrednio biblioteki *libmodbus* (na której bazuje *mbpoll*).

---

17. którego używaliśmy już m.in. do programowania mikrokontrolera STM32 i do wypisywania wiadomości "diagnostycznych" przez STM32

18. którego obsługa jest wymagana przez standard dla urządzeń zgodnych z Modbus i jest on dużo popularniejszy niż wariant ASCII

## 7.2.4 Modbus w STM32

Przykładową implementację Modbus RTU dla STM32 można znaleźć pod adresem: <https://github.com/opcode-eu-org-libs/STM32-ModbusRTU>.

## 8 Wykład wideo<sup>19</sup>

- *Układy programowalne* – <http://video.opcode.eu.org/09.04-programowalne.mkv>
- *Mikrokontrolery STM32* – <http://video.opcode.eu.org/09.05-stm32.mkv>

## 9 Lektury uzupełniające 🧐

- *Pinout płytki „blue pill”* (<https://stm32duino.com/forum/wordpress/wp-content/uploads/2016/07/The-Generic-STM32F103-Pinout-Diagram.pdf>) opis układu wyprowadzeń używanego w ramach zajęć modułu z mikrokontrolerem.
- *Karta katalogowa STM32F103* (<https://www.st.com/resource/en/datasheet/stm32f103c8.pdf>) opisująca pinout i parametry mikrokontrolera.
- *Reference manual* dla STM32F103 ([https://www.st.com/resource/en/reference\\_manual/cd00171190-stm32f101xx-stm32f102xx-stm32f103xx-stm32f105xx-and-stm32f107xx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf](https://www.st.com/resource/en/reference_manual/cd00171190-stm32f101xx-stm32f102xx-stm32f103xx-stm32f105xx-and-stm32f107xx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf)) – obszerny dokument opisujący w jaki sposób programować mikrokontroler. Zawiera szczegółowe opisy działania peryferiów, listę rejestrów i pól bitowych wraz ich funkcjami oraz adresami<sup>20</sup>. Najważniejszy dokument przy programowaniu mikrokontrolera.
- *Programming manual* dla STM32F103 ([https://www.st.com/resource/en/programming\\_manual/cd00228163-stm32f10xxx-20xxx-21xxx-l1xxxx-cortex-m3-programming-manual-stmicroelectronics.pdf](https://www.st.com/resource/en/programming_manual/cd00228163-stm32f10xxx-20xxx-21xxx-l1xxxx-cortex-m3-programming-manual-stmicroelectronics.pdf))
- *Dokumentacja libopencm3* (<http://libopencm3.org/docs/latest/html/>) opisująca funkcje i makra dostępne w bibliotece.
- *Przykładowy kod napisany z użyciem libopencm3* (<https://github.com/libopencm3/libopencm3-examples>)
- *Opis działania magistrali I<sup>2</sup>C wraz z formą i znaczeniem warunków START, STOP, itd* (<https://www.ti.com/lit/an/slva704/slva704.pdf>)
- *Hacker's Delight* - Henry S. Warren, Jr. - książka opisująca dużą ilość algorytmów realizowalnych za pomocą operacji bitowych.

---

19. Filmy posiadają napisy wgrane do kontenera multimedialnego jako osobny strumień – napisy mogą być włączone lub wyłączone w odtwarzaczu. W wielu filmach dużo dzieje się „na dole ekranu”, dlatego polecamy odtwarzać filmy z napisami umieszczonymi poniżej filmu, np. przy pomocy polecenia: `vlc --video-filter='croppadd{paddbottom=120}' --sub-margin=-10 PLIK.mkv`

20. W STM32 opis adresów rejestrów jest rozłożony pomiędzy *reference manual* i kartę katalogową. Informacje dotyczące programowania mikrokontrolera są w *Programming manual* (programowanie samego SCB - bloku głównego procesora) oraz *Reference manual* (programowanie poszczególnych peryferiów)

## 10 Zadania dodatkowe

### Zadanie 10.0.1

W programowaniu mikrokontrolerów często zachodzi potrzeba ustawienia pojedynczego bitu rejestru na 0 lub 1, albo zmiany jego wartości. Sprawdź (np. rozpisując ich działanie), które wyrażenie w rejestrze rejestr, na podstawie maski maska:

```
rejestr |= maska; // <- Wyrażenie 1
rejestr ^= maska; // <- Wyrażenie 2
rejestr &= ~maska; // <- Wyrażenie 3
```

- Ustawia te bity na 1,
- Ustawia te bity na 0,
- Odwraca wartości tych bitów.

### Zadanie 10.0.2

Jaką maską bitową można sprawdzić czy bit nr. 1 jest w stanie wysokim? A jaką można sprawdzić to samo, ale dla wszystkich bitów parzystych (na pozycjach 0, 2, 4 ...  $14^a$ )?

- a. 16 bitowa liczba ma bity "ponumerowane" od 0 do 15

### Zadanie 10.0.3

Napisz wyrażenia, które nie znając poprzedniej wartości 8-bitowego rejestru XYZZY, wykona operacje:

1. Ustawi jego drugi najmłodszy bit jako 1
2. Ustawi jego piąty najmłodszy bit jako 0
3. Odwróci jego najstarszy bit
4. Wyzeruje jego dolną połowę

*Wskazówka: Możesz wygenerować maskę z ustawionym bitem  $n$  za przesuwając jedynkę o  $n$  miejsc w lewo:  $(1 \ll n)$*