Python w Elektronicznej Sieci #13: Programowanie mikrokontrolerów STM32

Projekt „Matematyka dla Ciekawych Świata”,
Krzysztof Lasocki
<krz.lasocki@gmail.com>

2020-05-26

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. Można je sklonować do podkatalogu za pomocą:

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

Repozytorium zawiera link symboliczny `libopencm3` wskazujący na `../libopencm3-master`. Pliki wspomagające kompilację przykładów zakładają że w tym miejscu znajdą katalog z (skompilowaną wcześniej) biblioteką `libopencm3`. Jeżeli bibliotekę tę masz w innej lokalizacji zastąp ten link wskazującym na poprawną lokalizację katalogu z skompilowaną biblioteką.

1 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 `00_blink` repozytorium z przykładami. Plik ten ma następującą postać:

```c
#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 00_blink
make
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

Płytka BluePill której używamy posiada zworki kontrolujące tryb uruchomienia mikrokontrolera (piny `BOOT0` i `BOOT1`). Zworka, którą przestawiłeś/aś (`BOOT0`) na "1" podczas przygotowywania mikronotrolera do pracy powoduje uruchomienie w trybie bootloadera. Po zaprogramowaniu, `stm32flash` wydaje bootlaoderowi 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 znadujący sie w pamięci mikrokontrolera po jego zrestartowaniu (bez potrzeby ponownego wgrywania lub zmieniania ustawień zworek).

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 asemblerze), ale kod w ten sposób napisany będzie mniej czytelny i mniej przenośny. [1]

```
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[2](wywołaniem makra `rcc_periph_clock_enable` z odpowiednim parametrem)

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

1. Oraz niewiele szybszy.
2. 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

```
  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[3]. Następnie funkcja gpio_toggle zmienia stan pinu 13 w porcie C na przeciwny, co powoduje zapalenie lub zgaszenie LEDa.

Zadanie 1.0.1

Zmień ten program tak, aby dioda LED migała około dwa razy wolniej.

2 Obsługa wejść

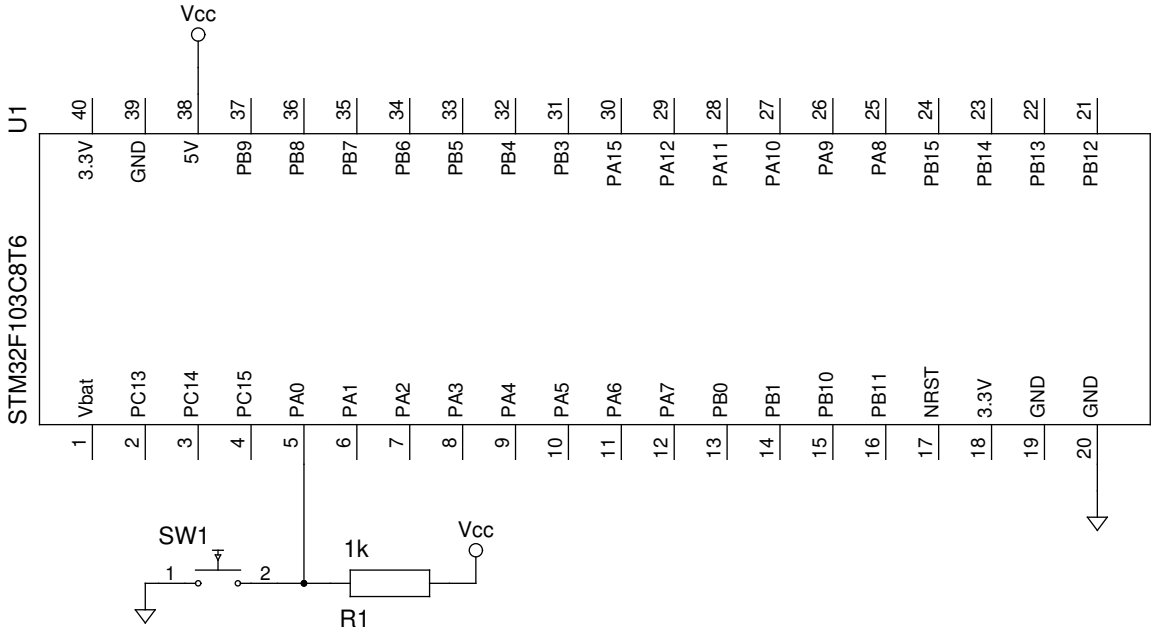
W tym ćwiczeniu pokażę, jak odczytywać stany logiczne pinów GPIO procesora oraz jak na ich podstawie podejmować decyzje. Skompiluj ten kod podobnie jak w poprzednim przykładzie (weź pod uwagę, że znajduje się on w innym katalogu: 01_di). Plik main.c wygląda następująco:

```
#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);
    }
```

3. W brew 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 odmierzania czasu.
Użycie wstawki asemblerowej - bo to oznacza __asm__() z instrukcją nop - zapobiega optymalizacji takiej pętli przez kompilator. Wstawek tego typu mozna też używać w celu umieszczenia w kodzie C lub C++ dowolnych innych instrukcji asemblerowych.

```
    // 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 skofigurować[4]. 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 nieustalone stany logiczne). Z tego powodu nieużywane wejścia z reguły podłącza się do znanego potencjału (GND lub napięcia zasilania układu) aby zapobiegać stanom nieustalonym. W mikrokontrolerach można też włączyć wbudowane rezystory podciągające. Pływającego wejścia można użyć jako źródła liczb losowych[5]

4. Wejście pływające (*Input, Input floating*) jest domyślną kofiguracją 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 `gpio_set` lub `gpio_clear`.
5. Jest to jedna z prostszych wersji takiego generatora, wcale nie pozbawiona innych wad

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.[6] 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):

```
  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 2.0.1

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 2.0.2

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

Zadanie 2.0.3

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

6. 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.

Zadanie 2.0.4

Zmodyfikuj ten program tak, aby dioda zapalała się tylko, gdy stan logiczny wejścia się zmienia. *Wskazówka: co zawiera zmienna* ***stan_a*** *tuż przed załadowaniem jej nowym stanem? Pamiętaj o stanach nieustalonych.*

3 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 `02_uart`, pierwszy z nich (`main.c`) ma postać:

```
#include <libopencm3/stm32/rcc.h>
#include <libopencm3/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 cudzysłowiach, 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[7]):

```
  gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
                GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);
```

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

Następnie konfigurowany jest sam UART. Tutaj ustawiamy typowe wartości - prędkość 9600 baud[8], 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 linuksa. 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;
}
```

Następnie przy pomocy pętli while, która wykona się `len` razy lub aż dojdzie do znaku zerowego w napisie[9], 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 potrzeby 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;
```

8. *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).
9. Prawdziwy `write` nie sprawdza tego drugiego warunku, ale ta wersja będzie służyć tylko do wypisywania napisów na UART

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
```

3.1 Odbieranie danych poprzez UART

W ostatnim ćwiczeniu nauczyliśmy się wysyłać dane poprzez interfejs UART. Uruchomiliśmy także wygodny[10] 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 przerwań.

10. Ta wygoda ma jednak swoją cenę. Jeżeli zauważyliście że ostatni program wgrywał 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.

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 przerwań, 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 `02b_uart2`. Pliki `uart.h` i `uart.c` mają postać dokładnie taką samą jak poprzednio i nie będziemy modyfikować 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 przerwań 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);
  // zmieniamy tryb na RX/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) {
  if ( USART_SR(USART1) & 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[11]:

```
  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 (UART1 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`:

```
  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:

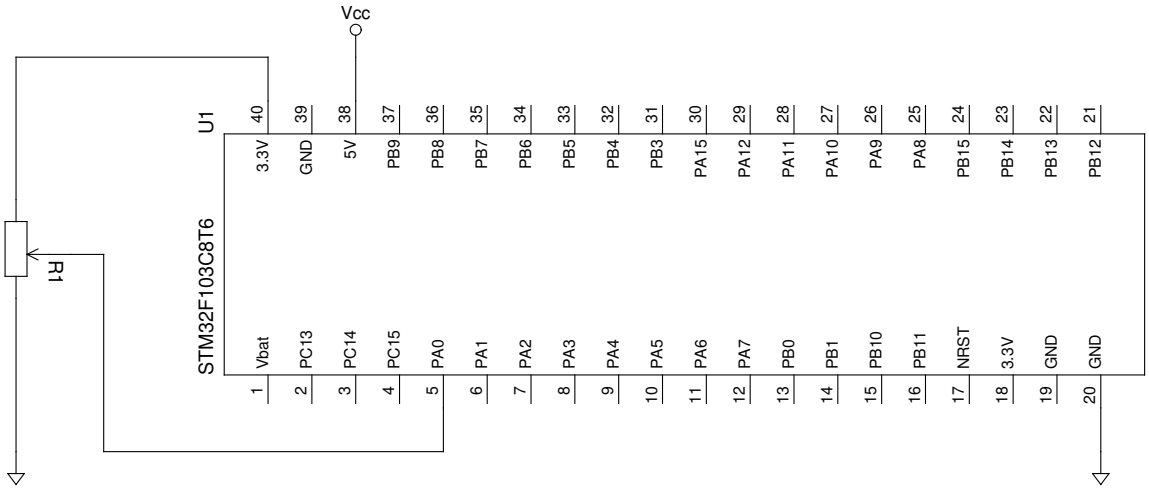
11. W tablicy tej, na dobrze zdefiniowanej pozycji (bajty 4-7, odpowiadające wyjątkowi *reset*), umieszczany jest także adres początku kodu związanego z naszym programem. To do tego adresu wykonywany jest skok gdy zrestartujemy nasz mikrokontroler z ustawioną zworkami BOOT opcją uruchamiania z pamięci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

```
  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);
    }
  }
```

4 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 `03_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 jedengo kanału. Pin A0 to kanał zerowy, więc umieszczemy zero w pierwszym elemencie tablicy którą przekażemy 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());
  }
}
```

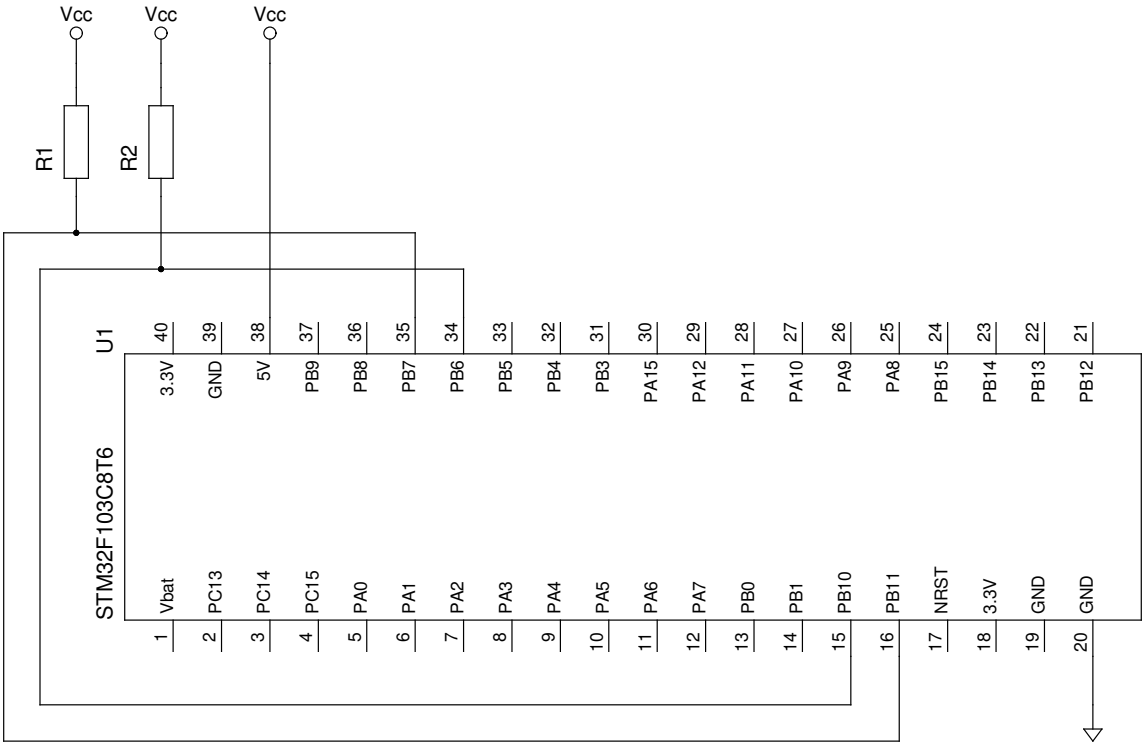
W `main` pojawiło się wywołanie `rcc_periph_clock_enable` uruchamiające zegar dla ADC, wywołanie naszej funkcji konfigurującej ADC oraz wypisanie wyniku konwersji w pętli za pomocą `printf()`.

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.

5 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[12] poprzez magistralę I^2C. Nasz układ wygląda tak:



Rezystory podciągające powinny mieć wartość kilku kΩ.

Kod do tego ćwiczenia znajduje się w katalogu `04_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);
```

12. 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.

```
  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);

  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_ITEVTEN );
  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_ITEVTEN )`, 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){

  uint32_t sr1, sr2;
  sr1 = I2C_SR1(I2C2);

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

  //Master write request
  else if(sr1 & I2C_SR1_RxNE){
    slavebyte = I2C_DR(I2C2);
    slavebyte *= 2;
    I2C_CR1(I2C2) = I2C_CR1(I2C2);
  }

  //Master read request
  if ((sr1 & I2C_SR1_TxE)){
    I2C_DR(I2C2) = slavebyte;
    I2C_CR1(I2C2) |= I2C_CR1_STOP;
  }
}
```

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 postawie 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 pewnie 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.

Jeśli poprzednia instrukcja warunkowa się nie wykonała, procedura sprawdza, czy w przypadku żądania odczytu przez mastera, 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 oraz dokonać zapisu do rejestru CR2. 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 CR2.[13]

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.

Należy potem zasygnalizować interfejsowi gotowość do wysłania danych, zapisując CR1[14].

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 wysłanie 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 wysłanie 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 wyslanie 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 pętle 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[15](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).

13. tutaj podobnie, interfejs czeka na tę operację i trzyma linię SCL w stanie niskim aż do jej wystąpienia, brak tej linijki spowoduje zawieszenie szyny
14. tak samo tutaj interfejs oczekuje tego zapisu do rejestru i będzie wymuszał stan niski na SCL dopóki taki zapis nie nastąpi.
15. Dokładna kolejność działań na rejestrach dla peryferium I^2C jest opisana w rozdziałach 26.3.2 (oprogramowanie slave'a) i 26.3.3 (oprogrwamowanie mastera) w *Reference manual*

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 wysłanie 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 wysłanie adresu
  while (!(I2C_SR1(peryf) & I2C_SR1_ADDR));
  (void) I2C_SR2(peryf); //Wyczyść EV6

  //Czekaj aż otrzymasz 1 bit 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 zapełnienie 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.

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

6 Lektury uzupełniające

- Tzw. *reference manual* dla STM32F103 (http://ln.opcode.eu.org/stm_rm0008) - obszerny dokument opisujący w jaki sposób programować mikrokontroler. Zawiera szczegółowe opisy działania peryferiów, listę rejestrów i pól bitowych wraz ich funkcjami oraz adresami [16]. Najważniejszy dokument przy programowaniu mikrokontrolera
- *Karta katalogowa STM32F103* (https://www.st.com/resource/en/datasheet/stm32f103c8.pdf) opisująca pinout i parametry mikrokontrolera.
- *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)
- *Vademecum informatyki praktycznej* (http://vip.opcode.eu.org/) - zbiór materiałów na temat elektroniki i programowania.
- *Hacker's Delight* - Henry S. Warren, Jr. - książka opisująca dużą ilość algorytmów realizowalnych za pomocą operacji bitowych.

7 Zadania

Zadanie 7.0.1

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`:

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

```
/* Wyrażenie 1 */
rejestr |= maska;

/* Wyrażenie 2 */
rejestr ^= maska;

/* Wyrażenie 3 */
rejestr &= ~maska;
```

16. 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)

Zadanie 7.0.2

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<<n)`

Zadanie 7.0.3

jaki sposób zmienić częstotliwość migania LEDa w pierwszym programie? Zmień ten program tak aby LED migał (około) dwa razy szybciej.

Zadanie 7.0.4

apisz program, który za pomocą zaimplementowanej w ćwiczeniu UART funkcji wejścia/wyjścia wypisze na UART "trójkąt z gwiazdek" jak poniżej

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

Zadanie 7.0.5

iedząc, że wartość 4096 odpowiada napięciu 3.3V, a 0 napięciu 0V, zmień przykładowy program 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 7.0.6

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