# Programowanie mikrokontrolerów 2.0

## Układy peryferyjne, wyjścia i wejścia binarne

Marcin Engel  Marcin Peczarski

Instytut Informatyki Uniwersytetu Warszawskiego

12 października 2021

# Co możemy podłączyć do mikrokontrolera?

Jako wyjście:

- diody świecące – LED (ang. *Light Emitting Diode*)
- wyświetlacz ciekłokrystaliczny – LCD (ang. *Liquid Crystal Display*)
- nadajnik podczerwieni – IR (ang. *Infra Red*)
- układ sterujący silniczkiem, serwomechanizmem, wiatraczkiem
- ...

# Co możemy podłączyć do mikrokontrolera?

Jako wejście:

- przycisk
- klawiaturę matrycową
- odbiornik IR
- termometr analogowy
- ...

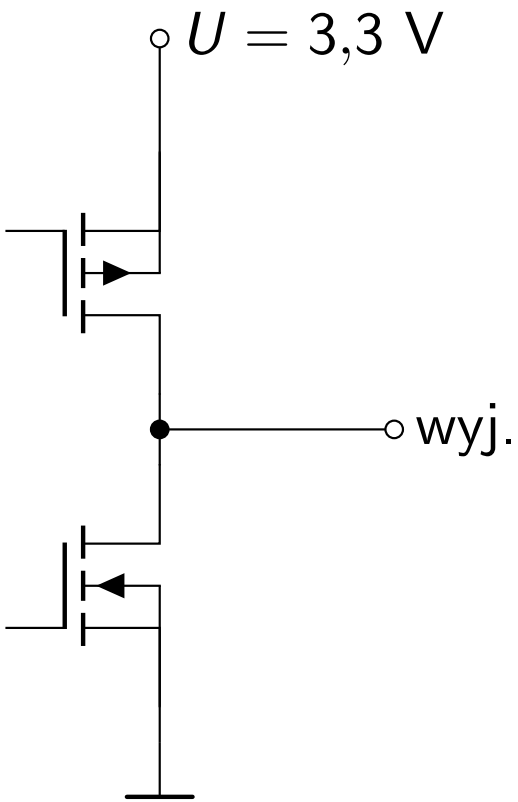
# Co możemy podłączyć do mikrokontrolera?

Jako wejście-wyjście:

- zewnętrzną pamięć
- komputer (np. łączem szeregowym)
- inny mikrokontroler
- akcelerometr
- inne czujniki
- ...

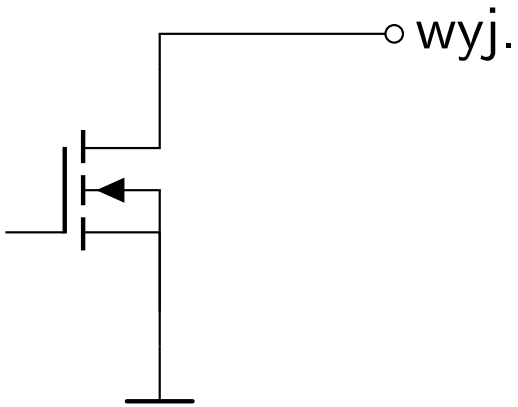
Będziemy poznawać sukcesywnie na kolejnych zajęciach

# Wyjście przeciwsobne (ang. *push-pull*)



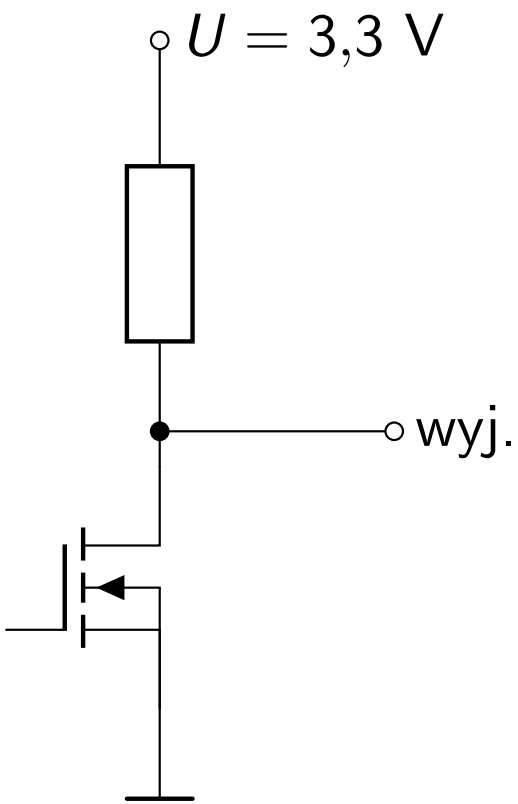

- Gdy zapiszemy na wyjście 0
  - dolny tranzystor przewodzi
  - górny tranzystor nie przewodzi
  - na wyjściu mikrokontrolera jest stan niski, napięcie bliskie 0 V – logiczne 0
- Gdy zapiszemy na wyjście 1
  - dolny tranzystor nie przewodzi
  - górny tranzystor przewodzi
  - na wyjściu mikrokontrolera jest stan wysoki, napięcie bliskie napięciu zasilania – logiczna 1
- Uwaga: tu i dalej stosujemy dodatnią konwencję logiczną

# Wyjście otwarty dren (ang. *open-drain*)




- Gdy zapiszemy na wyjście 0
  - tranzystor przewodzi
  - na wyjściu mikrokontrolera jest stan niski, napięcie bliskie 0 V – logiczne 0
- Gdy zapiszemy na wyjście 1
  - tranzystor nie przewodzi
  - na wyjściu mikrokontrolera jest stan nieustalony, czyli stan wysokiej impedancji – Z

# Wyjście otwarty dren z rezystorem podciągającym




- Gdy zapiszemy na wyjście 0
  - tranzystor przewodzi
  - na wyjściu mikrokontrolera jest stan niski, napięcie bliskie 0 V – logiczne 0
- Gdy zapiszemy na wyjście 1
  - tranzystor nie przewodzi
  - na wyjściu mikrokontrolera jest napięcie bliskie napięciu zasilania, słaby (ang. *weak*) stan wysoki – H
- Rezystor podciągający może być wewnętrzny lub zewnętrzny
- Uwaga: symbol H został wzięty z VHDL-a, ale w wielu dokumentach oznacza zwykły stan wysoki

# Wyjścia otwarty dren można ze sobą łączyć

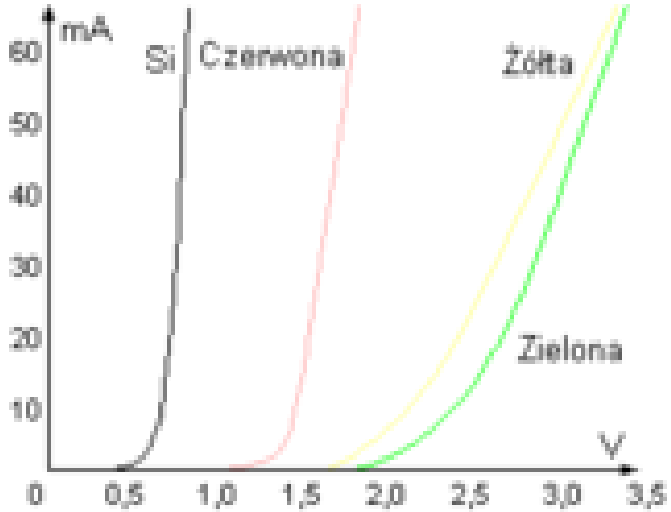
- Tworzą iloczyn logiczny

| $\wedge$ | 0 | H | Z |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| H | 0 | H | H |
| Z | 0 | H | Z |

- Aby uniknąć stanu nieustalonego, musi być przynajmniej jeden rezystor podciągający
- Stan wynikowy można odczytać, czytając wartość wejścia
- Korzysta się z tego np. w $I^2C$ i 1-Wire

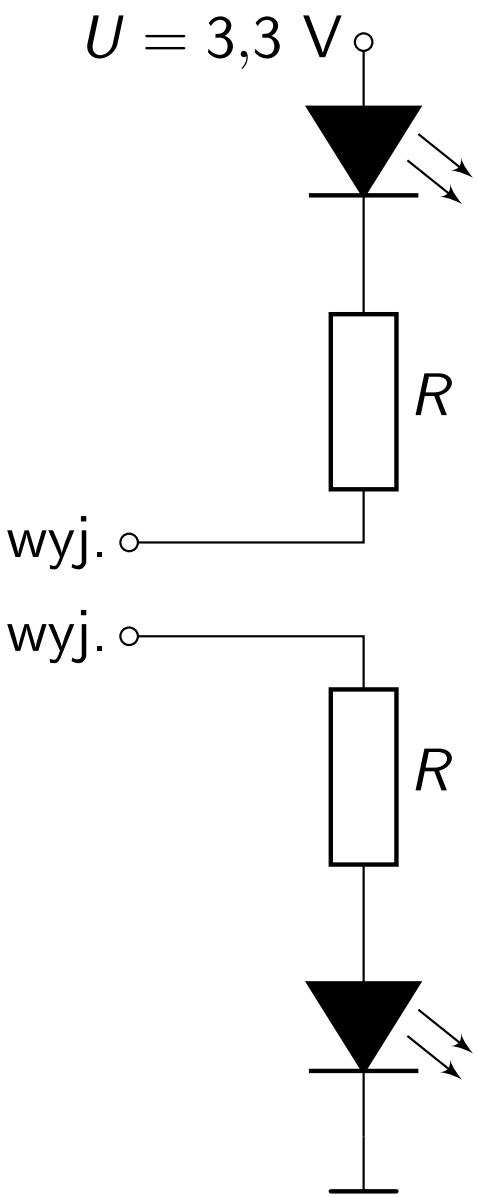
# Diody świecące, LED

# Charakterystyka diody




- Typowe parametry pracy
  - prąd od kilku do kilkudziesięciu mA
  - spadek napięcia od 0,2 V do 3,5 V
- Jasność diody świecącej zależy od prądu przez nią płynącego
- Żywotność też!
- Nadmierny prąd może uszkodzić diodę
- Dioda przewodzi prąd w jednym kierunku (kierunek przewodzenia)
- Zbyt duże napięcie przyłożone w kierunku zaporowym może uszkodzić diodę

# Jak przyłączyć diodę świecącą do mikrokontrolera?




- Prawo Ohma: $U = RI$
- Dla założonego prądu płynącego przez diodę:
  - sprawdzamy, jaki będzie spadek napięcia na niej
  - włączamy w obwód rezystor dobrany tak, aby spadki napięcia na nim i na diodzie sumowały się do napięcia zasilania
- Przykład:
  - prąd diody 5 mA
  - spadek napięcia na diodzie 1,7 V
  - napięcie zasilania 3,3 V
  - wartość rezystora:

$$\frac{3{,}3\ \text{V} - 1{,}7\ \text{V}}{5\ \text{mA}} = \frac{1{,}6\ \text{V}}{5\ \text{mA}} = 320\ \Omega$$

  - Wybieramy rezystor 330 Ω

# LED – podsumowanie

Dioda świecąca:

- przewodzi prąd w jednym kierunku (i wtedy świeci)
- wymaga ograniczenia prądu za pomocą rezystora

W zestawie laboratoryjnym:

- odpowiednie rezystory są wlutowane
- wyprowadzenie mikrokontrolera musi być ustawione jako wyjście przeciwsobne (ang. *push-pull*)
- po połączeniu diody z wyprowadzeniem mikrokontrolera dioda będzie świecić po podaniu na to wyprowadzenie stanu 0 lub 1 (zależnie od sposobu przyłączenia diody)

# Peryferie w STM32F411

- RCC
- GPIOA, GPIOB, GPIOC, GPIOD, GPIOE, GPIOH
- EXTI
- PWR, SYSCFG, FLASH
- ADC, ADC1
- TIM1, TIM2, TIM3, TIM4, TIM5, TIM9, TIM10, TIM11
- IWDG, WWDG
- RTC
- I2C1, I2C2, I2C3, I2S2ext, I2S3ext
- SDIO
- SPI1, SPI2, SPI3, SPI4, SPI5
- USART1, USART2, USART6
- USB_OTG_FS
- DMA1, DMA2
- CRC
- NVIC, SCB, SysTick
- Większość będziemy sukcesywnie poznawać

# Peryferie dostępne w innych STM32

- GPIOF, GPIOG, GPIOI, GPIOJ, GPIOK
- ADC2, ADC3
- DAC
- TIM6, TIM7, TIM8, TIM12, TIM13, TIM14
- CAN1, CAN2
- USART3, UART4, UART5, UART7, UART8
- SAI1
- SPI6
- USB_OTG_HS
- ETH
- LTDC, DCMI
- DMA2D
- FSMC, FMC
- CRYP, HASH, RNG
- MPU
- Nie będziemy o nich mówić, ale warto wiedzieć, że są dostępne w innych modelach STM32

# Jak korzysta się z peryferii?

- Programista widzi peryferie jako 16- lub 32-bitowe rejestry umieszczone w przestrzeni adresowej mikrokontrolera
- Przykład

```
uint16_t reg;
reg = *(volatile uint16_t*)(0x40020010);
*(volatile uint32_t*)(0x40020014) = 0;
```

- Co robi ten kod?
- Dlaczego użyto słowa kluczowego `volatile`?
- Nieeleganckie
- Podatne na trudne do wykrycia błędy

# Biblioteka CMSIS

- Cortex Microcontroller Software Interface Standard definiuje odpowiednie struktury, na przykład

```
#define __IO volatile
typedef struct {
  __IO uint32_t MODER;   /* mode                  */
  __IO uint32_t OTYPER;  /* output type           */
  __IO uint32_t OSPEEDR; /* output speed          */
  __IO uint32_t PUPDR;   /* pull-up/pull-down     */
  __IO uint32_t IDR;     /* input data            */
  __IO uint32_t ODR;     /* output data           */
  __IO uint32_t BSRR;    /* bit set/reset         */
  __IO uint32_t LCKR;    /* configuration lock    */
  __IO uint32_t AFR[2];  /* alternate function    */
} GPIO_TypeDef;
```

# Biblioteka CMSIS

- Dla każdego układu peryferyjnego CMSIS definiuje jego położenie w pamięci, na przykład

```
#define PERIPH_BASE                    (0x40000000)
#define AHB1PERIPH_BASE (PERIPH_BASE + 0x0020000)
#define GPIOA_BASE     (AHB1PERIPH_BASE + 0x0000)
#define GPIOA          ((GPIO_TypeDef *)GPIOA_BASE)
```

- Teraz kod może wyglądać czytelnie

```
uint16_t reg;
reg = GPIOA->IDR;
GPIOA->ODR = 0;
```

- Dostarcza też funkcji rozwijających się w miejscu do instrukcji asemblerowych realizujących operacje, których nie ma w języku C, np. `__NOP()`
- W labie dostępna w katalogu `/opt/arm/stm32/CMSIS`

# Biblioteka CMSIS

- Aby użyć CMSIS, należy przy wywoływaniu kompilatora
  - zdefiniować model procesora za pomocą stałej preprocesora
    `-DSTM32F411xE`
  - podać kompilatorowi, gdzie ma szukać plików nagłówkowych
    `-I/opt/arm/stm32/CMSIS/Include`
    `-I/opt/arm/stm32/CMSIS/Device/ST/STM32F4xx/Include`
  - włączyć właściwy plik nagłówkowy
    `#include <stm32f4xx.h>`
- Nazwa pliku nagłówkowego może się zmienić (w przeszłości już się to zdarzało)
- Może się okazać, że trzeba włączyć więcej plików
- Dobrze jest wszystko zgromadzić w jednym miejscu
  - sugerujemy włączanie pliku
    `#include <stm32.h>`
  - ścieżka dla kompilatora
    `-I/opt/arm/stm32/inc`
  - dzięki temu zmiana modelu mikrokontrolera nie musi wymagać modyfikowania kodu źródłowego, a tylko wystarczy go przekompilować

# Jak korzysta się z peryferii?

- Odczytać rejestr już umiemy

```
uint16_t reg;
reg = GPIOA->IDR;
```

- Zapis jest równie prosty

```
GPIOA->ODR = 0x00e0U;
```

- A co, gdy chcemy ustawić tylko niektóre bity?

```
GPIOA->ODR |= 0x00e0U;
```

- Albo wyzerować niektóre bity?

```
GPIOA->ODR &= ~0x00e0U;
```

- Albo zanegować niektóre bity?

```
GPIOA->ODR ^= 0x00e0U;
```

# Jak korzysta się z peryferii?

- Jak zmienić wartości bitów, których aktualnych wartości nie znamy?

```
uint32_t reg;
reg = GPIOA->MODER;
reg &= ~(3U << (2U * pin_n));
reg |= new_mode << (2U * pin_n);
GPIOA->MODER = reg;
```

- A właściwie dlaczego nie tak?

```
GPIOA->MODER &= ~(3U << (2U * pin_n));
GPIOA->MODER |= new_mode << (2U * pin_n);
```

- Pierwsza wersja generuje krótszy (wydajniejszy) kod maszynowy: tylko jeden odczyt i jeden zapis rejestru `MODER`
- Druga wersja skutkuje dwoma odczytami i zapisami oraz powoduje występowanie w rejestrze przez chwilę stanu pośredniego
- Dotychczas przedstawione sposoby modyfikowania bitów w rejestrach peryferyjnych nie są atomowe

# Atomowe modyfikowanie stanu wyjść

- Niektóre peryferie mają rejestry pozwalające na atomowe modyfikacje
- Bity w rejestrze `BSRR` mogą być tylko zapisywane
- Zapis bitu w młodszej połówce rejestru `BSRR` wartością 1 ustawia odpowiedni bit w rejestrze `ODR`
- Zapis bitu w starszej połówce rejestru `BSRR` wartością 1 zeruje odpowiedni bit w rejestrze `ODR`
- Zapisywanie wartości 0 do bitów rejestru `BSRR` jest ignorowane
- Ustaw na wyprowadzeniu `PA5` poziom wysoki

```
GPIOA->BSRR = 1U << 5;
```

- Ustaw na wyprowadzeniu `PA5` poziom niski

```
GPIOA->BSRR = 1U << (5 + 16);
```

# Taktowanie

- Zanim się czegokolwiek użyje, trzeba włączyć taktowanie
- Włączenie taktowania portów PA i PB (układy GPIOA i GPIOB)

```
RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN |
                RCC_AHB1ENR_GPIOBEN;
```

- Włączenie taktowania interfejsów USART1 i USART2

```
RCC->APB2ENR |= RCC_APB2ENR_USART1EN;
RCC->APB1ENR |= RCC_APB1ENR_USART2EN;
```

- Włączenie taktowania licznika TIM2

```
RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;
```

- Można dostrzec konwencję nazywania bitów:
  UKŁAD_REJESTR_BIT

# Diody świecące w zestawie laboratoryjnym

- Na płytce Nucleo mamy zieloną LED
  - przyłączoną do wyprowadzenia `PA5`
  - włączaną poziomem wysokim – logiczna 1
- Na ekspanderze mamy trójkolorową LED
  - czerwona – wyprowadzenie `PA6`
  - zielona – wyprowadzenie `PA7`
  - niebieska – wyprowadzenie `PB0`
  - włączane poziomem niskim – logiczne 0

# Hello World, czyli zamigaj diodami (1)

- Tworzymy plik `leds_main.c`
- Potrzebujemy różnych definicji

```
#include <delay.h>
#include <gpio.h>
#include <stm32.h>
```

- Pliki nagłówkowe są dostępne w labie w katalogu `/opt/arm/stm32/inc`

# Hello World, czyli zamigaj diodami (2)

- Należy zdefiniować konfigurację sprzętu

```
#define RED_LED_GPIO    GPIOA
#define GREEN_LED_GPIO  GPIOA
#define BLUE_LED_GPIO   GPIOB
#define GREEN2_LED_GPIO GPIOA

#define RED_LED_PIN    6
#define GREEN_LED_PIN  7
#define BLUE_LED_PIN   0
#define GREEN2_LED_PIN 5
```

# Hello World, czyli zamigaj diodami (3)

- Dobrze jest zdefiniować powtarzające się operacje, zwiększając jednocześnie czytelność kodu

```
#define RedLEDon()     \
  RED_LED_GPIO->BSRR = 1 << (RED_LED_PIN + 16)
#define RedLEDoff()    \
  RED_LED_GPIO->BSRR = 1 << RED_LED_PIN

...

#define Green2LEDon()  \
  GREEN2_LED_GPIO->BSRR = 1 << GREEN2_LED_PIN
#define Green2LEDoff() \
  GREEN2_LED_GPIO->BSRR = 1 << (GREEN2_LED_PIN + 16)
```

- Czy ktoś widzi różnicę?

# Hello World, czyli zamigaj diodami (4)

```
int main() {
  RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN |
                  RCC_AHB1ENR_GPIOBEN;
  __NOP();

  RedLEDoff();
  GreenLEDoff();
  BlueLEDoff();
  Green2LEDoff();
```

# Hello World, czyli zamigaj diodami (5)

```
GPIOoutConfigure(RED_LED_GPIO,
                 RED_LED_PIN,
                 GPIO_OType_PP,
                 GPIO_Low_Speed,
                 GPIO_PuPd_NOPULL);


...

GPIOoutConfigure(GREEN2_LED_GPIO,
                 GREEN2_LED_PIN,
                 GPIO_OType_PP,
                 GPIO_Low_Speed,
                 GPIO_PuPd_NOPULL);
```

# Hello World, czyli zamigaj diodami (6)

```
    for (;;) {
      RedLEDon();
      Delay(4000000);
      RedLEDoff();
      GreenLEDon();
      Delay(4000000);
      GreenLEDoff();
      BlueLEDon();
      Delay(4000000);
      BlueLEDoff();
      Green2LEDon();
      Delay(4000000);
      Green2LEDoff();
    }
  }
```

# Pytania

- Dlaczego wyłączamy diody przed skonfigurowaniem wyprowadzeń?
- Po co jest instrukcja `NOP`?
- Jakie wartości parametrów przyjmuje funkcja `GPIOoutConfigure`?

# Jak skompilować? Plik `makefile`

```
CC       = arm-eabi-gcc
OBJCOPY  = arm-eabi-objcopy
FLAGS    = -mthumb -mcpu=cortex-m4
CPPFLAGS = -DSTM32F411xE
CFLAGS   = $(FLAGS) -Wall -g \
           -O2 -ffunction-sections -fdata-sections \
           -I/opt/arm/stm32/inc \
           -I/opt/arm/stm32/CMSIS/Include \
  -I/opt/arm/stm32/CMSIS/Device/ST/STM32F4xx/Include
LDFLAGS  = $(FLAGS) -Wl,--gc-sections -nostartfiles \
           -L/opt/arm/stm32/lds -Tstm32f411re.lds

vpath %.c /opt/arm/stm32/src
```

# Jak skompilować? Plik makefile, cd.

```
OBJECTS = leds_main.o startup_stm32.o delay.o gpio.o
TARGET  = leds

.SECONDARY: $(TARGET).elf $(OBJECTS)

all: $(TARGET).bin

%.elf : $(OBJECTS)
        $(CC) $(LDFLAGS) $^ -o $@

%.bin : %.elf
        $(OBJCOPY) $< $@ -O binary

clean :
        rm -f *.bin *.elf *.hex *.d *.o *.bak *~
```

# Optymalizacja

- Mikrokontrolery mają ograniczoną pamięć i ograniczoną częstotliwość taktowania
- Optymalizacje przyspieszające wykonywanie kodu zwykle zmniejszają też jego rozmiar
- W większości przypadków najodpowiedniejsza jest opcja `-O2`
- Zmniejszenie rozmiaru kodu wykonywalnego można uzyskać, stosując opcje `-ffunction-sections` i `-fdata-sections`
  - biblioteki muszą być skompilowane z tymi opcjami
  - konsolidator musi być wywołany z opcją `--gc-sections`
- Podglądanie efektu pracy kompilatora:
  `arm-eabi-objdump -M reg-names-std -d *.elf`

# Jak zaprogramować mikrokontroler?

Podłączyć, znajdujący się na płytce Nucleo, ST-LINK/V2-1 do gniazda USB w komputerze

Pierwszy sposób (jeśli nie potrzebujemy debugera)

- Użyć OpenOCD w trybie wsadowym (w bieżącym katalogu musi być obraz pamięci `*.bin`) za pomocą skryptu `/opt/arm/stm32/ocd/qfn4`

Drugi sposób (jeśli potrzebujemy debuger)

- Uruchomić OpenOCD jako GDB-serwer za pomocą skryptu `/opt/arm/stm32/ocd/dbgn4` – w katalogu domowym musi być plik `.gdbinit` zawierający coś takiego

```
target remote :3333
monitor reset halt
```

- W katalogu, w którym mamy plik wykonywalny, uruchomić GDB poleceniem `arm-eabi-gdb *.elf`
- Następnie w GDB użyć polecenia `load`

# Jak zaprogramować mikrokontroler?

Trzeci sposób (bez OpenOCD)

- ST-LINK/V2-1 jest widoczny jako pamięć masowa USB, np. jako dysk `/dev/sdb`
- Można go gdzieś podmontować, np. do katalogu `/run/media/user/NUCLEO`
- i skopiować obraz pamięci, np. tak:

```
cp leds.bin /run/media/user/NUCLEO
```

# Jak zaimplementować prostą funkcję opóźniającą?

- Pierwszy pomysł

```
void Delay(unsigned count) {
  while (count--);
}
```

- Dlaczego `count--`, a nie `--count`?

```
Delay(0);
```

- Dlaczego ta implementacja jest zła?

```
00000000 <Delay>:
  0:   4770        bx      lr
  2:   bf00        nop
```

# Jak zaimplementować prostą funkcję opóźniającą?

- Użyjmy słowa kluczowego `volatile`, żeby zablokować optymalizację

```
void Delay(volatile unsigned count) {
  while (count--);
}
```

- Dlaczego ta implementacja jest zła?

```
00000000 <Delay>:
   0:   b082        sub     sp, #8
   2:   9001        str     r0, [sp, #4]
   4:   9b01        ldr     r3, [sp, #4]
   6:   1e5a        subs    r2, r3, #1
   8:   9201        str     r2, [sp, #4]
   a:   2b00        cmp     r3, #0
   c:   d1fa        bne.n   4 <Delay+0x4>
   e:   b002        add     sp, #8
  10:   4770        bx      lr
  12:   bf00        nop
```

# Jak zaimplementować prostą funkcję opóźniającą?

- Spróbujmy inaczej

```
void Delay(unsigned count) {
  while (count--) {
    __NOP();
    __NOP();
  }
}
```

- Dlaczego ta implementacja jest dobra?

```
00000000 <Delay>:
   0:   1e43        subs    r3, r0, #1
   2:   b120        cbz     r0, e <Delay+0xe>
   4:   bf00        nop
   6:   bf00        nop
   8:   f113 33ff   adds.w  r3, r3, #4294967295 ; -1
   c:   d2fa        bcs.n   4 <Delay+0x4>
   e:   4770        bx      lr
```

# Jak zaimplementować prostą funkcję opóźniającą?

- Nasza ostateczna wersja funkcji opóźniającej

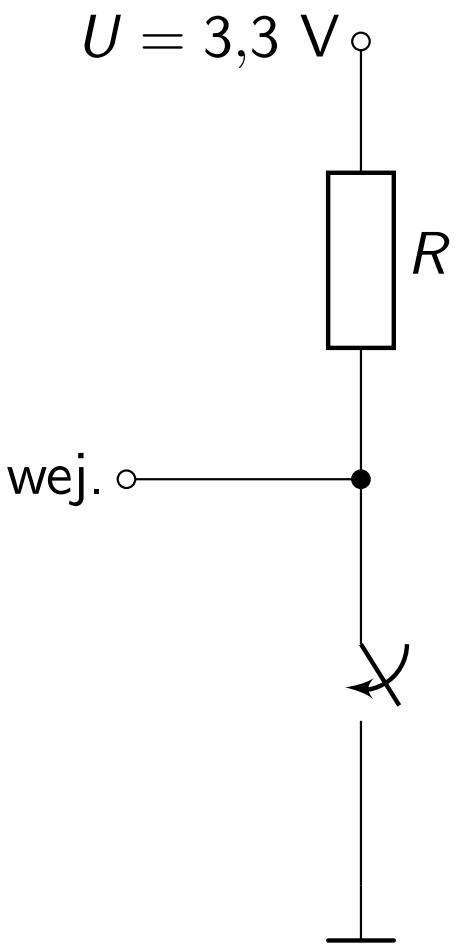
```
void Delay(unsigned count) {
  while (count--) {
    __NOP();
    __NOP();
  }
}
```

- Dlaczego dwie instrukcje `NOP`?
- Jak dobrać wartość argumentu, aby otrzymać żądane opóźnienie?

$$250tf$$

gdzie $t$ to żądany czas opóźnienia w ms, a $f$ to częstotliwość taktowania mikrokontrolera w MHz

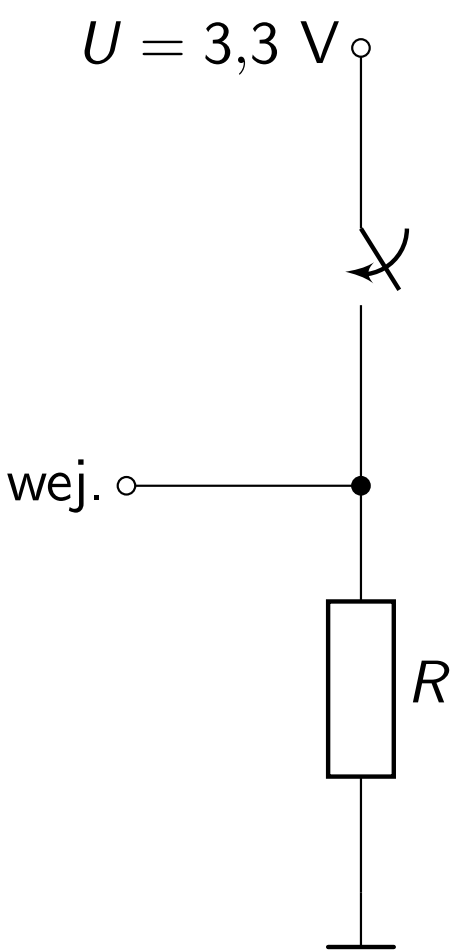
# Jak przyłączyć przycisk do mikrokontrolera?




- Gdy przycisk niewciśnięty, rezystor ustala na wejściu mikrokontrolera stan wysoki, napięcie bliskie napięciu zasilania – logiczna 1
- Wciśnięcie przycisku ustala na wejściu mikrokontrolera stan niski, napięcie bliskie 0 V – logiczne 0
- Gdy przycisk niewciśnięty, przez rezystor prąd praktycznie nie płynie
- Gdy przycisk wciśnięty, przez rezystor płynie prąd, przykładowo

$$I = \frac{U}{R} = \frac{3{,}3\ \text{V}}{10\ \text{k}\Omega} = 0{,}33\ \text{mA}$$

# Jak przyłączyć przycisk do mikrokontrolera?




- Gdy przycisk niewciśnięty, rezystor ustala na wejściu mikrokontrolera stan niski, napięcie bliskie 0 V – logiczne 0
- Wciśnięcie przycisku ustala na wejściu mikrokontrolera stan wysoki, napięcie bliskie napięciu zasilania – logiczna 1
- Gdy przycisk niewciśnięty, przez rezystor prąd praktycznie nie płynie
- Gdy przycisk wciśnięty, przez rezystor płynie prąd, przykładowo

$$I = \frac{U}{R} = \frac{3{,}3\ \text{V}}{10\ \text{k}\Omega} = 0{,}33\ \text{mA}$$

# Zjawisko drgania styków

- W rzeczywistości wciśnięcie i puszczenie przycisku powoduje mikrodrgania

- Stan stabilizuje się po pewnym czasie

# Algorytm obsługi przycisku bez powtarzania

```
if (przycisk wciśnięty) then
begin
  wait(T);
  if (przycisk wciśnięty) then
  begin
    obsłuż zdarzenie;
    while (przycisk wciśnięty)
      wait(T)
  end
end
```

# Algorytm obsługi przycisku z powtarzaniem

```
if (przycisk wciśnięty) then
begin
  wait(T);
  while (przycisk wciśnięty) then
  begin
    obsłuż zdarzenie;
    count := 0;
    while (przycisk wciśnięty and (count < timeout))
    begin
      wait(T);
      count := count + 1
    end;
    zmodyfikuj(timeout)
  end
end
```

# Przycisk – podsumowanie

Przycisk monostabilny zwierny:

- przewodzi prąd, gdy jest wciśnięty
- wymaga rezystora ustalającego stan, gdy jest niewciśnięty
- stabilizuje swój stan po kilku do kilkunastu milisekundach od wciśnięcia lub puszczenia

W zestawie laboratoryjnym:

- wyprowadzenie mikrokontrolera musi być ustawione jako wejście – tak jest ustawione po włączeniu zasilania lub wyzerowaniu mikrokontrolera
- odpowiednie zewnętrzne rezystory podciągające do zasilania lub ściągające do masy są wlutowane (ustawienie wewnętrznego rezystora nie przeszkadza)

# Przyciski w zestawie laboratoryjnym

- Na płytce Nucleo mamy przycisk USER
  - przyłączony do wyprowadzenia `PC13`
  - stan aktywny niski – logiczne 0
- Na ekspanderze mamy dżojstik
  - przycisk w lewo – wyprowadzenie `PB3`
  - przycisk w prawo – wyprowadzenie `PB4`
  - przycisk w górę – wyprowadzenie `PB5`
  - przycisk w dół – wyprowadzenie `PB6`
  - przycisk akcja – wyprowadzenie `PB10`
  - stan aktywny niski – logiczne 0
- Na ekspanderze mamy przycisk AT MODE
  - przyłączony do wyprowadzenia `PA0`
  - stan aktywny wysoki – logiczna 1

# Sprawdzanie stanu przycisku

- Należy zdefiniować konfigurację sprzętu, przykładowo

```
#define UP_BTN_GPIO GPIOB
#define UP_BTN_PIN  5
#define AT_BTN_GPIO GPIOA
#define AT_BTN_PIN  0
```

- Przycisk „w górę” jest wciśnięty, gdy fałszywy jest warunek (wyrażenie ma wartość zero)

```
(UP_BTN_GPIO->IDR >> UP_BTN_PIN) & 1
```

- Przycisk „AT MODE” jest wciśnięty, gdy prawdziwy jest warunek (wyrażenie ma wartość jeden)

```
(AT_BTN_GPIO->IDR >> AT_BTN_PIN) & 1
```

# Układy GPIO – podsumowanie

- Mikrokontroler ma kilka układów `GPIO` (ang. *General Purpose Input Optput*) – portów wejścia wyjścia
- Są oznaczane kolejnymi literami alfabetu
  - układy wejścia-wyjścia: `GPIOA`, `GPIOB`, `GPIOC`, ...
  - porty wejścia-wyjścia: `PA`, `PB`, `PC`, ...
- Każdy port obsługuje 16 wyprowadzeń (wejść-wyjść), oznaczanych kolejnymi liczbami: `PA0`, `PA1`, ..., `PA15`, `PB0`, `PB1`, ..., `PB15`, `PC0`, `PC1`, ..., `PC15`, ...
- Odczyt stanu wejścia odbywa się poprzez rejestr `IDR`
- Ustawienie stanu wyjścia odbywa się poprzez rejestr `ODR` albo rejestr `BSRR`
  - ustawiony stan wyjścia można odczytać z rejestru `ODR`
  - rzeczywisty stan wyjścia można odczytać z rejestru `IDR`

# Konfigurowanie GPIO – podsumowanie

- Każde wyprowadzenie może być indywidualnie konfigurowane jako
  - wejście cyfrowe
  - wyjście cyfrowe
  - funkcja alternatywna
  - wejście analogowe
- Po włączeniu zasilania lub wyzerowaniu mikrokontrolera większość wyprowadzeń jest skonfigurowana jako wejście cyfrowe
- Patrz pliki `gpio.h`, `gpio.c` dostępne w labach w katalogach odpowiednio `/opt/arm/stm32/inc`, `/opt/arm/stm32/src`