Od wejścia do komunikacji
Arduino UNO R4 WiFi w praktyce

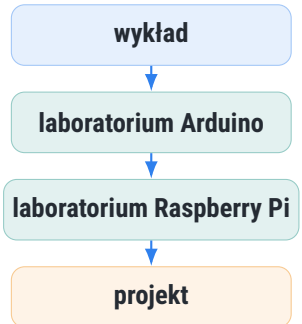
GPIO, czas, przerwania, PWM, UART i I^2C

Podstawy Systemów Wbudowanych • Wykład 1
dr inż. Angelika Tefelska



Organizacja przedmiotu

- **6 wykładów po 2 h + 7. wykład jednogodzinny**, co dwa tygodnie,
- laboratoria odbywają się co tydzień,
- każdy dwugodzinny wykład przygotowuje do **dwóch kolejnych laboratoriów**,
- na wykładzie skupiamy się na zasadach, decyzjach projektowych i czytaniu kodu,
- na laboratorium: uruchomienie, pomiary, diagnostyka i rozbudowa programu.



Regulamin przedmiotu

- Ocena z zajęć laboratoryjnych jest wystawiana w następujący sposób:
  - 3,0-3,5 - w przypadku zaliczenia wszystkich ćwiczeń laboratoryjnych oraz miniprojektu,
  - +1,5 stopnia - w przypadku wykonania projektu końcowego.
- Ocena z wykładu jest wystawiana na podstawie kolokwium zgodnie z poniższą tabelą:

Ocena	Wynik procentowy z kolokwium
5,0	[91;100]%
4,5	[81;91)%
4,0	[71;81)%
3,5	[61;71)%
3,0	[51;61)%

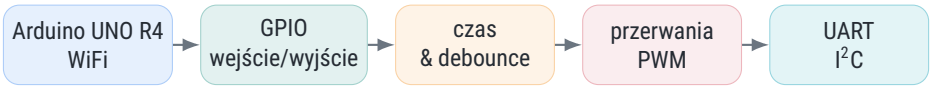
Ocena	Ocena końcowa
5,0	[4,5; 5.0]
4,5	[4.25; 4.5)
4,0	[3.75; 4.25)
3,5	[3.25; 3.75)
3,0	[3.0; 3.25)

- Ocena końcowa jest równa średniej ważonej ocen z laboratorium i wykładu z wagami odpowiednio: 2/3 i 1/3 odpowiednio (przy czym obydwie oceny muszą być pozytywne). Ocena końcowa jest wystawiana zgodnie z powyższą tabelą.
- Materiały do wykładu i laboratoriów: `https://labe.engined.eu`.

Plan 7 wykładów

1. **Od wejścia do komunikacji** — Arduino UNO R4 WiFi, GPIO, czas, przerwania, PWM, UART, I^2C.
2. **Od czujnika do aktuatora** — ADC, czujniki, SPI, 1-Wire, serwo, silnik, przekaźnik, drivery.
3. **Arduino w Internecie** — Wi-Fi, IP, HTTP, JSON, MQTT i podstawy bezpieczeństwa IoT.
4. **Od mikrokontrolera do Raspberry Pi** — MCU vs SBC, przypomnienie Linuxa i Pythona, GPIO na Raspberry Pi 5.
5. **Raspberry Pi jako węzeł i brama IoT** — I^2C/SPI/UART, czujniki, MQTT, gateway, edge computing.
6. **Kamera i analiza obrazu** — Picamera2, obraz cyfrowy, OpenCV, progowanie, kontury i detekcja.
7. **Od modułów do kompletnego systemu** (1 h) — integracja, testowanie, dokumentacja i architektura projektu.

Mapa dzisiejszego wykładu

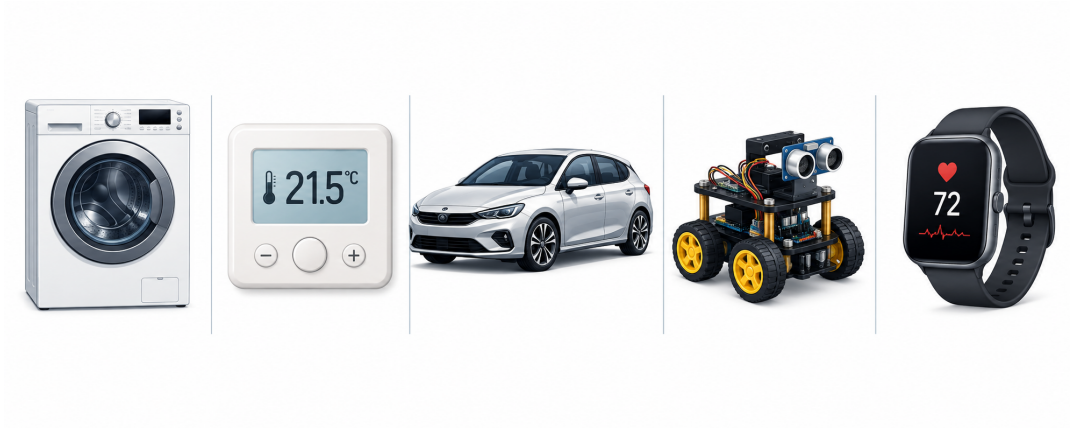


Cel: patrząc na fragment kodu umieć odpowiedzieć
co robi sprzęt, kiedy program czeka, a kiedy reaguje na zdarzenie.

1. System wbudowany i Arduino

Zaczynamy od platformy Arduino UNO R4 WiFi.

Komputery są wszędzie — tylko nie zawsze wyglądają jak komputer

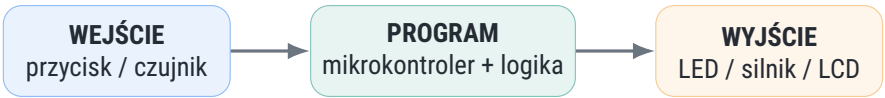


- pralka, termostat, samochód, robot i smartwatch mają **program, wejścia i wyjścia**,
- różnią się zastosowaniem, ale wszystkie **zbierają dane i sterują sprzętem**.

System wbudowany: część większego urządzenia

Definicja robocza

System wbudowany to mikrokontroler/komputer będący **częścią większego urządzenia**, zaprojektowany do realizacji określonych funkcji i współpracy ze sprzętem: czujnikami, przyciskami, napędami, wyświetlaczami lub siecią.



Najprostszy schemat myślenia: **odczytaj świat** → **podejmij decyzję** → **wykonaj działanie**.

System wbudowany ma ograniczenia

- ograniczona pamięć i moc obliczeniowa,
- ograniczony pobór energii,
- określone napięcia i prądy,
- czasem wymagania czasowe,
- praca bez klawiatury, ekranu i użytkownika,
- konieczność działania przez długi czas.

Konsekwencja

Kod „działający” to za mało. Musimy odpowiedzieć sobie na pytania:

- czy nie blokuje innych zadań?
- co stanie się po błędzie?
- czy wejście ma ustalony poziom?
- czy pin nie jest przeciążony?
- czy będzie pracował wystarczająco długo na zaprojektowanym zasilaniu (baterie/powerbank/zasilacz)?

Mikrokontroler a komputer jednopłytkowy

	Mikrokontroler	SBC/SoC, np. Raspberry Pi
Start	program bezpośrednio po resecie	system operacyjny, potem aplikacja
Pamięć	mała, wbudowana	dużo większa, RAM + pamięć masowa
Czas reakcji	przewidywalniejszy	zależny m.in. od systemu operacyjnego
GPIO	bezpośrednia praca z peryferiami	GPIO przez sterowniki/system
Typowe użycie	sterowanie, pomiar, automatyka	sieć, GUI, kamera, analiza danych

W PSW użyjemy obu podejść: najpierw Arduino UNO R4 WiFi, później Raspberry Pi 5.

Skąd wzięło się Arduino? — krótka historia

- projekt powstał około **2005 r.** w Interaction Design Institute Ivrea we Włoszech,
- celem było stworzenie **taniej i łatwej w użyciu** platformy dla studentów sztuk pięknych ,
- w rozwój byli zaangażowani m.in. **Massimo Banzi, David Cuartielles i David Mellis**,
- sprzęt i środowisko programistyczne miały pozwalać szybko przejść od pomysłu do działającego prototypu,
- nazwa *Arduino* nawiązuje do baru w Ivrei, w którym spotykali się twórcy projektu.


CZY

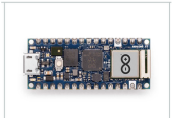
?

Jednym z założeń była cena porównywalna z prostym posiłkiem, a nie drogim zestawem laboratoryjnym.

Arduino to cała rodzina płytek



Arduino UNO R4 WiFi



Arduino Nano RP2040 Connect



Arduino Mega 2560 Rev3



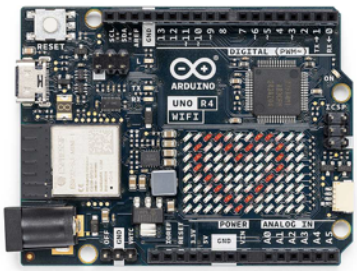
Arduino MKR WiFi 1010

- **UNO** – uniwersalne płytki edukacyjne i prototypowe,
- **Nano** – małe płytki do kompaktowych projektów,
- **Mega / GIGA** – więcej wyprowadzeń i zasobów,
- **MKR** – płytki nastawione m.in. na komunikację i IoT.

Źródło: https://www.arduino.cc/en/hardware

Co dostajemy jako platformę?

- płytkę z mikrokontrolerem **Renesas RA4M1** (Arm Cortex-M4, 48 MHz),
- moduł **ESP32-S3** do Wi-Fi i Bluetooth LE,
- piny GPIO, ADC/DAC i interfejsy UART, I^2C, SPI, CAN,
- matrycę LED 12×8, USB-C i złącze Qwiic,
- Arduino IDE / CLI, Arduino Core, biblioteki i przykłady.



Arduino UNO R4 WiFi

Źródło: https://www.arduino.cc/en/hardware

GPIO i magistrale — dwa sposoby wykorzystania wyprowadzeń

GPIO — General Purpose Input/Output

- uniwersalne wyprowadzenie mikrokontrolera,
- może być **wejściem** lub **wyjściem**,
- służy np. odczytu stanu przycisku, zapalenia diody LED itd.,
- stan pinu może być LOW lub HIGH.

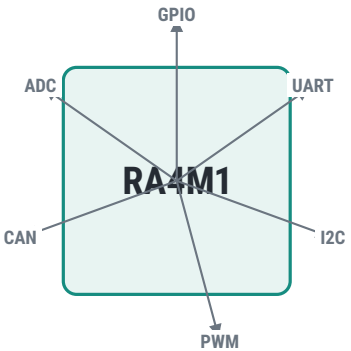
Magistrala / interfejs komunikacyjny

- kilka linii pracuje według określonych reguł transmisji,
- przykłady: **UART, I²C, SPI**,
- ten sam fizyczny pin może mieć **funkcję alternatywną**.

Dlatego w pinoucie jednego pinu często widzimy kilka różnych nazw.

RA4M1 — to on steruje większością naszych pinów

- 32-bitowy rdzeń Arm Cortex-M4 z FPU,
- 256 kB Flash i 32 kB SRAM,
- GPIO i timery,
- ADC i DAC,
- UART, I^2C, SPI, CAN,
- RTC, OPAMP i inne peryferia.



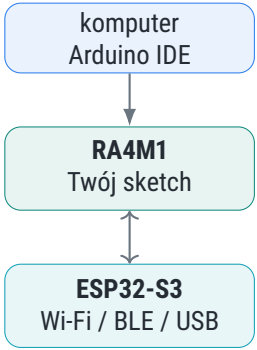
Dziś

Interesują nas głównie **GPIO, timer, przerwania, PWM, UART i I^2C**.

Źródło: https://docs-content.arduino.cc/resources/datasheets/ABX00087-datasheet.pdf

Dwa układy na jednej płytce: RA4M1 i ESP32-S3

- **RA4M1** wykonuje typowy sketch użytkownika i obsługuje większość wyprowadzeń Arduino,
- **ESP32-S3** odpowiada przede wszystkim za Wi-Fi, Bluetooth LE i elementy komunikacji USB,
- układy komunikują się wewnętrznie — w zwykłym programie nie traktujemy ich jako dwóch niezależnych „Arduino”,
- główne GPIO RA4M1 pracują z logiką **5 V**,
- sygnały dostępne na **złączu ESP32-S3** mają poziom **3.3 V** i nie są odporne na 5 V.



Źródło: https://docs.arduino.cc/hardware/uno-r4-wifi

Pin to fizyczne wyprowadzenie — obowiązują maksymalne napięcia i prądy

Przed podłączeniem czegokolwiek sprawdź

1. jakie **napięcie** pojawi się na pinie,
2. jaki **prąd** może popłynąć,
3. czy pin pracuje jako **wejście**, **wyjście** czy funkcja magistrali.

- główne GPIO Arduino UNO R4 WiFi mają poziomy logiczne **5 V**,
- maksymalny prąd pojedynczego GPIO: **8 mA**,
- do głównych GPIO nie podajemy napięć większych niż 5 V,
- złącze ESP32-S3 i Qwiic pracują przy **3.3 V**.



Źródło: https://store.arduino.cc/products/uno-r4-wifi

Arduino UNO R4 WiFi — najważniejsze wyprowadzenia

UART
D0/RX, D1/TX

I²C
A4/SDA, A5/SCL

SPI
D10–D13

PWM
D3,D5,D6,D9,D10,D11

Zasilanie i wejścia analogowe

IOREF / RESET
+3.3 V / +5 V / GND / VIN

A0 / D14
DAC + wejście analogowe

A1 / D15
wejście analogowe

A2 / D16
wejście analogowe

A3 / D17
wejście analogowe

A4 / D18
SDA — I²C

A5 / D19
SCL — I²C



Piny cyfrowe D0–D13

D0 / RX
UART RX

D1 / TX
UART TX

D2
GPIO + przerwanie

D3
PWM + przerwanie

D5, D6
PWM

D9, D10, D11
PWM

D10–D13
SPI: SS/COPI/CIPO/SCK

D13
LED_BUILTIN + SCK

Ważne elektrycznie
Główne GPIO RA4M1: logika 5 V, maks. 8 mA na pin. Złącze ESP32-S3 i Qwiic: 3.3 V.

Który układ wykonuje typowy sketch Arduino na Arduino UNO R4 WiFi?

1. ESP32-S3,
2. RA4M1,
3. oba zawsze wykonują ten sam program równolegle,
4. komputer podłączony przez USB.

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

Który układ wykonuje typowy sketch Arduino na Arduino UNO R4 WiFi?

1. ESP32-S3,
2. **RA4M1,** - typowy sketch użytkownika jest wykonywany przez **Renesas RA4M1**. ESP32-S3 pełni na tej płytce przede wszystkim rolę układu łączności i elementu toru USB. To ważne także elektrycznie: główne GPIO RA4M1 są 5-woltowe, natomiast ESP32-S3 pracuje przy 3.3 V.
3. oba zawsze wykonują ten sam program równolegle,
4. komputer podłączony przez USB.

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

2. GPIO — wejście i wyjście cyfrowe

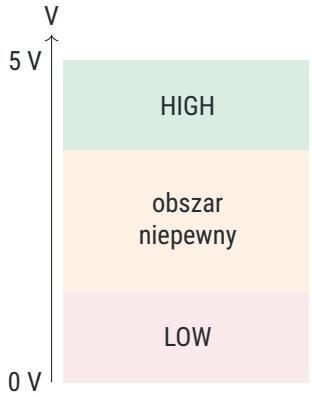
Najpierw: dioda LED. Potem: przycisk i wejście z rezystorem podciągającym.

Sygnał cyfrowy nie jest abstrakcyjnym 0 i 1

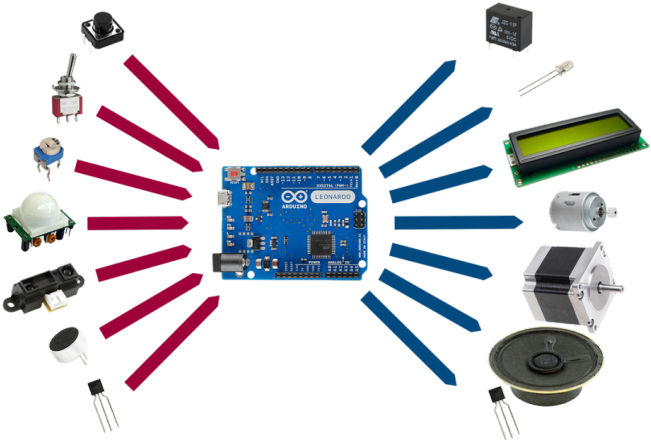
- mikrokontroler mierzy **napięcie**,
- odpowiednio niskie napięcie interpretuje jako LOW,
- odpowiednio wysokie napięcie interpretuje jako HIGH,
- obszar pomiędzy progami jest niepożądany.

Wniosek

Wejście powinno mieć **zawsze określony stan elektryczny**.



Elementy wejściowe i wyjściowe

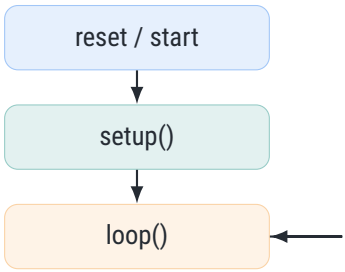


- **wejścia** dostarczają informacji do mikrokontrolera: przycisk, czujnik, odbiornik komunikacji,
- **wyjścia** pozwalają mikrokontrolerowi oddziaływać na otoczenie: LED, przekaźnik, silnik, nadajnik.

Źródło: https://blog.codebender.cc/2014/03/07/lesson-1-inputs-and-outputs/

Struktura programu Arduino: setup() i loop()

```
1  void setup() {
2    // wykonuje sie raz po starcie / resecie
3  }
4
5  void loop() {
6    // wykonuje sie wielokrotnie
7    // tak dlugo, jak dziala program
8  }
```



Pierwszy program: migająca dioda

```
const int LED = LED_BUILTIN;

void setup() {
  pinMode(LED, OUTPUT);
}

void loop() {
  digitalWrite(LED, HIGH);
  delay(500);
  digitalWrite(LED, LOW);
  delay(500);
}
```

Co już widać?

Konfigurujemy kierunek pinu przez `pinMode()`, a stan wyjścia zmieniamy przez `digitalWrite()`.

pinMode() — konfiguracja kierunku i podciągania

- `pinMode(pin, OUTPUT)` — pin jest **wyjściem**; program może wymuszać LOW/HIGH,
- `pinMode(pin, INPUT)` — pin jest **wejściem** bez wewnętrznego podciągania,
- `pinMode(pin, INPUT_PULLUP)` — wejście z **wewnętrznym rezystorem podciągającym** do zasilania.

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  pinMode(2, INPUT_PULLUP);
}
```

Praktyczna wskazówka

Dla prostego przycisku podłączonego między pin i GND bardzo często wybieramy `INPUT_PULLUP`.

digitalWrite() i digitalRead()

Najpierw konfigurujemy pin przez `pinMode()`, a dopiero potem go odczytujemy lub ustawiamy.

Wyjście

```
digitalWrite(pin, HIGH)
```

```
digitalWrite(pin, LOW)
```

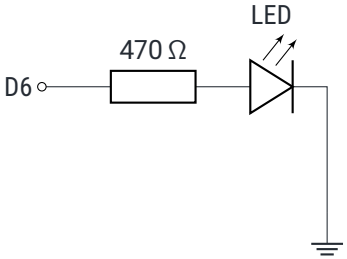
Program ustala stan logiczny pinu skonfigurowanego jako wyjście.

Wejście

```
int stan = digitalRead(pin);
```

Funkcja zwraca `HIGH` albo `LOW` zgodnie z poziomem logicznym na wejściu.

Zewnętrzna LED: dobieramy rezystor ograniczający prąd



Dla czerwonej LED przyjmijmy spadek napięcia około **1.7 V**.

470 Ω – bezpieczny przykład

$$I \approx \frac{5 - 1.7}{470} \approx 7.0\,mA$$

To wartość poniżej limitu 8 mA GPIO.

A rezystor 330 Ω?

$$I \approx \frac{5 - 1.7}{330} \approx 10\,mA$$

Dla Arduino UNO R4 WiFi **nie polecamy** – przekracza specyfikowany maksymalny prąd 8 mA na pin.

Uwaga na tutoriale dla starych płytek Arduino, które pokazują podłączenie diody LED przez rezystor 220-330Ω.

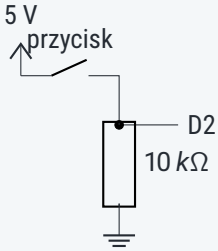
Wejście nie może „pływać” — pull-down i pull-up

Wejście bez podciągania

- wysoka impedancja,
- łatwo zbiera zakłócenia,
- odczyt może losowo zmieniać HIGH/LOW.

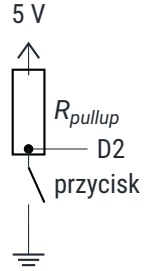
Nie zostawiamy wejścia „w powietrzu”.

Pull-down



spoczynek = LOW
naciśnięcie = HIGH

Pull-up

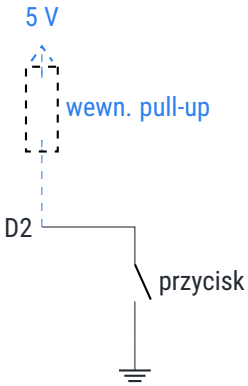


spoczynek = HIGH
naciśnięcie = LOW

Dlaczego często wybieramy pull-up?

Arduino może włączyć rezystor wewnętrzny przez `INPUT_PULLUP`, więc prosty przycisk wymaga tylko połączenia z GND.

INPUT_PULLUP — przycisk steruje diodą



- przycisk puszczony: HIGH,
- przycisk naciśnięty: LOW.

```arduino
const int BUTTON = 2;
const int LED = LED_BUILTIN;

void setup() {
  pinMode(BUTTON, INPUT_PULLUP);
  pinMode(LED, OUTPUT);
}

void loop() {
  if (digitalRead(BUTTON) == LOW)
  {
    digitalWrite(LED, HIGH);
  }
  else
  {
        digitalWrite(LED, LOW);
  }
}
```

Odwrócona logika nie jest błędem — zmienna `pressed`

	przycisk puszczony	przycisk naciśnięty
INPUT_PULLUP	HIGH	LOW
zmienna `pressed`	false	true

```
const int BUTTON = 2;
const int LED = LED_BUILTIN;

void setup() {
  pinMode(BUTTON, INPUT_PULLUP);
  pinMode(LED, OUTPUT);
}

void loop() {
  bool pressed = (digitalRead(BUTTON) == LOW);

  if (pressed) {
    digitalWrite(LED, HIGH);
  } else {
    digitalWrite(LED, LOW);
  }
}
```

Stan a zdarzenie: to nie to samo

- **Stan:** „przycisk jest teraz naciśnięty” — interesuje nas bieżący poziom.
- **Zdarzenie:** „przycisk właśnie został naciśnięty” — porównujemy stan bieżący z poprzednim.

```
const int BUTTON = 2;
int previous = HIGH;  // stan z poprzedniego obiegu petli

void loop() {
  int current = digitalRead(BUTTON);  // stan teraz

  if (current == LOW && previous == HIGH) {
    // przejscie HIGH -> LOW = jedno nowe nacisniecie
  }

  previous = current;  // stan dla kolejnego obiegu
}
```

W `setup()` pin D2 konfigurujemy wcześniej jako `INPUT_PULLUP`.

Przycisk jest podłączony między D2 i GND, a D2 ma INPUT_PULLUP. Jaki stan odczytamy po naciśnięciu?

1. HIGH,
2. LOW,
3. losowy — zawsze,
4. zależy od `digitalWrite()`.

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

Przycisk jest podłączony między D2 i GND, a D2 ma INPUT_PULLUP. Jaki stan odczytamy po naciśnięciu?

1. HIGH,
2. **LOW,** - Wewnętrzny rezystor utrzymuje wejście w stanie HIGH, kiedy przycisk jest otwarty. Po naciśnięciu przycisk łączy pin z GND, więc odczytujemy LOW. Ta „odwrócona logika” jest normalna i bardzo często spotykana w systemach wbudowanych.
3. losowy — zawsze,
4. zależy od `digitalWrite()`.

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

3. Czas bez blokowania

`delay()` jest proste, ale zatrzymuje cały program. Poznajmy `millis()`.

Co jest nie tak z delay()?

```
1 digitalWrite(LED, HIGH);
2 delay(5000);       // przez 5 s program czeka
3 digitalWrite(LED, LOW);
```

- w czasie `delay()` kolejne instrukcje w `loop()` nie są wykonywane,
- program może nie sprawdzić przycisku wtedy, gdy użytkownik go naciska jeśli taki fragment kodu byłby po zgaszeniu diody LED,
- może opóźnić odczyt czujnika albo obsługę danych przychodzących,
- im więcej zadań ma aplikacja, tym trudniej budować ją z kolejnych blokujących opóźnień.

Wniosek

`delay()` jest przydatne w prostych demonstracjach, ale większy system wymaga zwykle **czekania bez blokowania**.

millis() — sprawdzamy czas bez zatrzymywania programu

- `millis()` zwraca liczbę milisekund od startu programu,
- samo wywołanie funkcji **nie zatrzymuje** programu,
- zapamiętujemy czas ostatniej akcji i sprawdzamy, czy minął wymagany odstęp.

```
1  const unsigned long interval = 500; // czas miedzy akcjami [ms]
2  unsigned long previous = 0;         // czas poprzedniej akcji
3
4  unsigned long now = millis();
5
6  if (now - previous >= interval) {
7    previous = now;
8    // wykonaj akcje raz na interval ms
9  }
```

Jak rozumieć powyższy kod?

„Jeśli od poprzedniego wykonania minęło co najmniej `interval` ms — wykonaj akcję i zapamiętaj nowy czas.”

Blink bez delay()

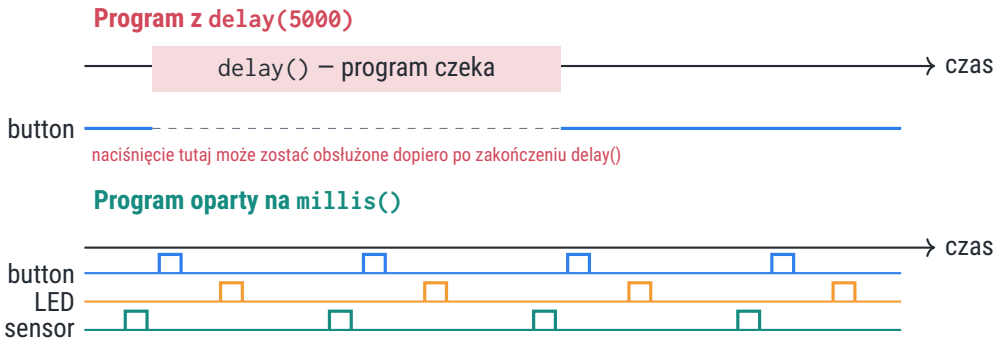
```
const int LED = LED_BUILTIN;
const unsigned long INTERVAL = 500;
unsigned long previous = 0;
bool ledState = false;

void setup() {
  pinMode(LED, OUTPUT);
}

void loop() {
  unsigned long now = millis();

  if (now - previous >= INTERVAL) {
    previous = now;
    ledState = !ledState;
    digitalWrite(LED, ledState);
  }
}
```

Dlaczego warto unikać blokowania? — dwa przebiegi czasowe



Pętla stale wraca do wszystkich zadań; każde z nich tylko sprawdza, czy nadszedł jego moment.

Kilka niezależnych zadań w jednej pętli

```
unsigned long tLed = 0;
unsigned long tSensor = 0;

void loop() {
  unsigned long now = millis();

  if (now - tLed >= 500) {
    tLed = now;
    toggleLed();
  }

  if (now - tSensor >= 2000) {
    tSensor = now;
    readSensor();
  }

  checkButton();
}
```

Uwaga: `toggleLed()`, `readSensor()` i `checkButton()` to **nasze własne funkcje**. Ich definicje pominięto, żeby pokazać wyłącznie organizację czasu w `loop()`.

A co, gdy millis() się przepełni?

- licznik `millis()` nie rośnie w nieskończoność,
- po pewnym czasie zawija się do zera,
- dlatego nie piszemy logiki opartej na „absolutnym czasie docelowym”,
- odejmowanie liczb bez znaku działa poprawnie także przez moment zawinięcia.

Dobry wzorzec

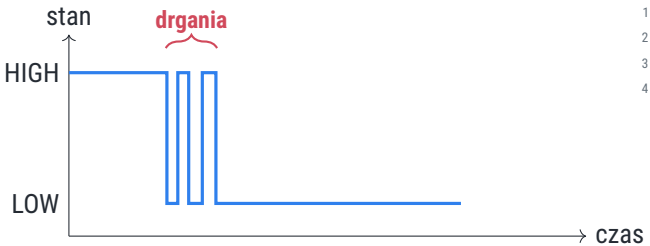
```
if (now - previous >= interval)
```

Unikaj

```
if (now >= previous + interval)
```

To dobry przykład różnicy między kodem, który działa „teraz”, a kodem odpornym na długą pracę.

Drgania styków — jeden nacisk może wyglądać jak kilka




```
if (current == LOW &&
    previous == HIGH) {
  counter++;
}
```

Skutek

Kilka przejść HIGH→LOW może dać kilka inkrementacji, mimo jednego fizycznego naciśnięcia.

Rozwiązanie: **debounce** — akceptujemy zmianę dopiero po krótkim czasie stabilności.

Najprostszy debounce z delay() — działa, ale blokuje

```arduino
if (digitalRead(BUTTON) == LOW) {
  delay(30);
  if (digitalRead(BUTTON) == LOW) {
    // uznajemy nacisniecie
  }
}
```

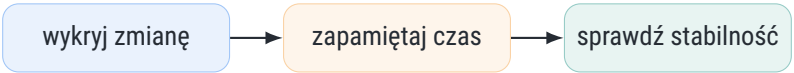
Zaleta

Łatwy do zrozumienia i często wystarczy w bardzo prostym programie.

Wada

Program przez ten czas nie obsługuje innych zadań w `loop()`.

Lepsza idea: zaakceptuj zmianę dopiero, gdy jest stabilna



Jeśli od ostatniej surowej zmiany minęło np. **30 ms** i wejście nadal ma ten sam stan — dopiero wtedy aktualizujemy „pewny” stan przycisku.

Debounce bez blokowania — wersja praktyczna

```
const int BUTTON = 2;
const unsigned long DEBOUNCE_MS = 30;
int lastRaw = HIGH;
int stableState = HIGH;
unsigned long lastChange = 0;

void loop() {
  int raw = digitalRead(BUTTON);
  unsigned long now = millis();

  if (raw != lastRaw) {
    lastRaw = raw;
    lastChange = now;
  }

  if (now - lastChange >= DEBOUNCE_MS && raw != stableState) {
    stableState = raw;
    if (stableState == LOW) {
      // jedno pewne nacisniecie
    }
  }
}
```

Krótki nacisk, długi nacisk, podwójny klik — wszystko opiera się na czasie

Moment naciśnięcia

Zapamiętaj `tPress = millis()`.

Moment puszczenia

Policz `duration = millis() - tPress`.

Czas	Interpretacja
< 500 ms	krótki klik
500–1500 ms	dłuższe przytrzymanie
> 1500 ms	długi nacisk

Progi są decyzją projektową — program powinien być napisany tak, by łatwo je zmienić.

Dlaczego w większym programie częściej wybierzemy millis() niż delay()?

1. bo `millis()` działa szybciej elektrycznie,
2. bo pozwala sprawdzać upływ czasu bez zatrzymywania całej pętli,
3. bo `delay()` nie działa na Arduino UNO R4 WiFi,
4. bo `millis()` automatycznie usuwa drgania styków.

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

Dlaczego w większym programie częściej wybierzemy millis() niż delay()?

1. bo `millis()` działa szybciej elektrycznie,
2. **bo pozwala sprawdzać upływ czasu bez zatrzymywania całej pętli,**
3. bo `delay()` nie działa na Arduino UNO R4 WiFi,
4. bo `millis()` automatycznie usuwa drgania styków.

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

4. Przerwania

Reakcja na zdarzenie: sprzęt może sam zgłosić ważną zmianę.

Polling a przerwanie — analogia

Polling

Czytasz książkę, ale co kilka sekund sam sprawdzasz telefon:

„czy ktoś dzwoni?”
„czy ktoś dzwoni?”
„czy ktoś dzwoni?”

Tak działa np. regularne `digitalRead()` w `loop()`.

Przerwanie

Czytasz książkę spokojnie. Gdy ktoś zadzwoni, **telefon sam przerywa** Ci na chwilę czytanie.

„DZWONEK — TERAZ!”

Procesor przerywa normalny kod, wykonuje krótką ISR i wraca.

Kiedy warto użyć przerwania?

- gdy impuls może być **krótki** i łatwo go przeoczyć w wolnej pętli — np. impuls z enkodera,
- gdy zdarzenie wymaga szybkiej reakcji — np. sprzętowy sygnał alarmowy z czujnika wykrycia pożaru,
- gdy peryferium sprzętowe samo zgłasza zakończenie operacji — timer, odbiór danych, pomiar,
- gdy chcemy zliczać impulsy niezależnie od chwilowego obciążenia głównej pętli.

Nie zawsze jest potrzebne

Dla zwykłego przycisku odczytywanego w szybkiej pętli polling + debounce jest często prostszy i wystarczający.

Na naszych ćwiczeniach

- D2 — przerwanie,
- D3 — przerwanie + PWM,
- używamy `digitalPinToInterrupt(pin)`.

Źródło: https://docs.arduino.cc/tutorials/uno-r4-wifi/cheat-sheet/

attachInterrupt() — składnia

```
attachInterrupt(
  digitalPinToInterrupt(BUTTON),
  buttonISR,
  FALLING
);
```

źródło
który pin?

ISR
jaką funkcję uruchomić?

tryb
przy jakiej zmianie?

Tryby wyzwalania przerwania

Tryb	Kiedy wywołuje ISR?
RISING	przy przejściu LOW $\rightarrow$ HIGH
FALLING	przy przejściu HIGH $\rightarrow$ LOW
CHANGE	przy każdej zmianie stanu
LOW	gdy wejście pozostaje w stanie LOW — użycie zależy od platformy/API

Przycisk z INPUT_PULLUP

Naciśnięcie daje przejście HIGH $\rightarrow$ LOW, więc naturalnym wyborem jest `FALLING`.

ISR powinna być krótka

Dobrze

- ustaw flagę,
- zwiększ prosty licznik,
- zapisz minimalną informację o zdarzeniu,
- wróć do programu.

Źle

- długie obliczenia,
- oczekiwanie,
- rozbudowany protokół komunikacyjny,
- logika całej aplikacji w ISR.

ISR sygnalizuje zdarzenie. Główna pętla je obsługuje.

volatile — informacja dla kompilatora o zmiennej współdzielonej z ISR

```
volatile bool buttonEvent = false;

void buttonISR() {
  buttonEvent = true;
}
```

- normalnie kompilator może założyć, że zmienna zmienia się tylko w miejscach widocznych w aktualnie wykonywanym kodzie,
- ISR może zmienić wartość **w dowolnym momencie** pomiędzy kolejnymi instrukcjami głównego programu,
- słowo `volatile` mówi: „za każdym razem odczytaj tę wartość z pamięci — nie zakładaj, że nadal jest taka sama”.

Minimalny przykład przerwania

```
const int BUTTON = 2;
volatile bool event = false;

void buttonISR() {
  event = true;
}

void setup() {
  pinMode(BUTTON, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(BUTTON),buttonISR, FALLING);
}

void loop() {
  if (event) {
    event = false;
    // obsluga zdarzenia poza ISR
  }
}
```

Bezpieczniejszy wzorzec: ISR ustawia flagę, loop() robi resztę

```
volatile bool irqFlag = false;

void buttonISR() {
  irqFlag = true;
}

void loop() {
  if (irqFlag) {
    noInterrupts();
    irqFlag = false;
    //krotkie polecenia
    interrupts();

    // tutaj mozna wykonac dluzsza obsluge
    // np. aktualizowac stan aplikacji
  }
}
```

Dla pojedynczej flagi ten przykład pokazuje przede wszystkim zasadę: **minimalna ISR, właściwa praca w loop()**.

Przycisk i przerwanie: nadal istnieją drgania styków

Przerwanie nie jest filtrem debounce

Jeśli przycisk odbije kilka razy, może wygenerować kilka przerwań.

Podejście A

ISR tylko ustawia flagę, a `loop()` wykonuje debounce na podstawie czasu.

Podejście B

Dla zwykłego przycisku rezygnujemy z przerwania i robimy polling + debounce.

Prostsza architektura często jest lepsza niż „więcej mechanizmów”.

Co najlepiej zrobić w ISR obsługującej naciśnięcie przycisku?

1. zrobić `delay(50)` i wypisać komunikat,
2. ustawić flagę zdarzenia i szybko wrócić,
3. poczekać, aż przycisk zostanie puszczony.

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

Co najlepiej zrobić w ISR obsługującej naciśnięcie przycisku?

1. zrobić `delay(50)` i wypisać komunikat,
2. **Pustawić flagę zdarzenia i szybko wrócić,** - Przerwanie ma umożliwić szybkie zanotowanie zdarzenia. Dłuższa logika powinna zostać wykonana później w `loop()`. Dodatkowo zwykły mechaniczny przycisk nadal wymaga rozwiązania problemu drgań styków.
3. poczekać, aż przycisk zostanie puszczony.

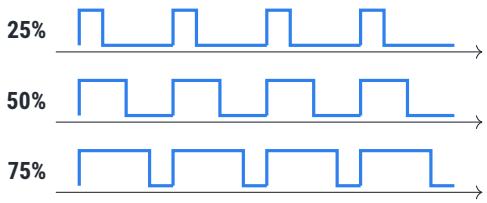
?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

5. PWM

Sterowanie współczynnikiem wypełnienia sygnału cyfrowego.

PWM — Pulse Width Modulation i współczynnik wypełnienia



każdy przebieg ma ten sam okres, ale inny czas HIGH

$$D = \frac{t_{HIGH}}{T} \cdot 100\%$$

Gdzie korzystamy z PWM?

- ”regulacja” obserwowanej jasności LED — oko uśrednia szybkie miganie,
- sterowanie mocą silnika DC przez odpowiedni driver,
- serwomechanizm.

Częstotliwość i wypełnienie opisują różne rzeczy

Częstotliwość f

Jak często powtarza się okres sygnału?

$$f = \frac{1}{T}$$

Wypełnienie D

Jaką część każdego okresu sygnał pozostaje w stanie HIGH?

$$D = \frac{t_{HIGH}}{T}$$

Nie myl tych parametrów

Dwa sygnały mogą mieć tę samą częstotliwość, ale różne wypełnienie — albo odwrotnie.

Oficjalne piny PWM na Arduino UNO R4 WiFi

D3	D5	D6	D9	D10	D11
PWM	PWM	PWM	PWM	PWM	PWM

Używamy pinów oznaczonych na płytce symbolem **~**. Nie zakładamy, że każdy GPIO ma tę samą funkcję PWM.

Źródło: https://docs.arduino.cc/tutorials/uno-r4-wifi/cheat-sheet/

analogWrite() — podstawowe sterowanie PWM

```
const int LED_PWM = 6;

void setup() {
  pinMode(LED_PWM, OUTPUT);
}

void loop() {
  analogWrite(LED_PWM, 64);   // ok. 25% przy 8 bitach
  delay(1000);
  analogWrite(LED_PWM, 128);  // ok. 50%
  delay(1000);
  analogWrite(LED_PWM, 255);  // ok. 100%
  delay(1000);
}
```

Domyślna rozdzielczość `analogWrite()` na Arduino UNO R4 WiFi to 8 bitów: 0…255.

Płynne rozjaśnianie LED

```
const int LED_PWM = 6;

void setup() {
  pinMode(LED_PWM, OUTPUT);
}

void loop() {
  for (int value = 0; value <= 255; value++) {
    analogWrite(LED_PWM, value);
    delay(5);
  }

  for (int value = 255; value >= 0; value--) {
    analogWrite(LED_PWM, value);
    delay(5);
  }
}
```

Na laboratorium zrobimy lepiej

To dobry pokaz PWM, ale dwa `for()` z `delay()` nadal blokują resztę programu.

Arduino UNO R4 pozwala zwiększyć rozdzielczość PWM

```ino
void setup() {
  analogWriteResolution(12);
}

void loop() {
  analogWrite(6, 2048);  // ok. 50% z zakresu 0..4095
}
```

	8 bitów	12 bitów
Liczba poziomów	256 (0–255)	4096 (0–4095)
Krok wypełnienia	ok. 0.39%	ok. 0.024%
Kiedy wybrać?	LED, proste sterowanie, gdy 256 kroków wystarcza	gdy potrzebne są bardzo małe zmiany i płynniejsze sterowanie

Więcej bitów nie oznacza „większego napięcia”

Rozdzielczość określa tylko, jak drobno możemy ustawić wypełnienie PWM. Nie zwiększa zakresu napięcia pinu.

PWM to nie jest prawdziwe napięcie analogowe

Na pinie PWM

W każdej chwili sygnał ma poziom LOW albo HIGH. Zmienia się tylko proporcja czasu.

Dlaczego LED wygląda na ciemniejszą?

Oko i sam układ optyczny uśredniają szybkie przełączanie, więc odbieramy zmianę jasności.

Gdy naprawdę potrzebujesz napięcia analogowego

Arduino UNO R4 WiFi ma także sprzętowy DAC na A0 — ale to temat Wykładu 2.

Co robi analogWrite(6, 128) przy domyślnej rozdzielczości?

1. ustawia na D6 stałe 2.5 V,
2. generuje PWM o wypełnieniu około 50%,
3. odczytuje napięcie 128 mV,
4. ustawia częstotliwość 128 Hz.

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

Co robi analogWrite(6, 128) przy domyślnej rozdzielczości?

1. ustawia na D6 stałe 2.5 V,
2. **generuje PWM o wypełnieniu około 50%,**
3. odczytuje napięcie 128 mV,
4. ustawia częstotliwość 128 Hz.

?

Najpierw spróbuj odpowiedzieć bez zaglądania do kodu.

6. Komunikacja — UART i I^2C

Drugie laboratorium: GPS mówi po UART, a wyświetlacz LCD współpracuje przez I^2C.

Czym jest magistrala komunikacyjna?

Magistrala

Zestaw linii sygnałowych oraz reguł określających **kto, kiedy i w jaki sposób przesyła bity** między układami.

Szeregowa asynchroniczna

UART
brak linii zegara; strony ustalają tempo transmisji.

Szeregowa synchroniczna

I^2C, SPI
osobna linia zegara synchronizuje transmisję.

Inne

1-Wire, CAN
specjalizowane rozwiązania do innych zastosowań.

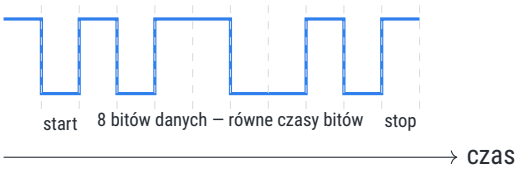
Wybór magistrali zależy od liczby urządzeń, wymaganej szybkości, liczby przewodów i odległości.

Którą magistralę wybrać? — zalety, wady i typowe zastosowania

Interfejs	Zalety	Ograniczenia	Typowe zastosowanie
UART	prosty, pełny dupleks, łatwa diagnostyka	zwykle punkt–punkt, obie strony muszą znać baud rate	GPS, modem, komunikacja z drugim mikrokontrolerem
I^2C	tylko 2 linie, wiele adresowanych urządzeń	wymaga pull-upów, adresy mogą się konfliktować, raczej krótkie połączenia	LCD, sensory, RTC
SPI	bardzo szybki, pełny dupleks	więcej przewodów, zwykle osobny CS dla każdego urządzenia	pamięć, RFID, szybkie ADC/DAC
1-Wire	jedna linia danych, urządzenia mają unikalne identyfikatory	wolniejszy, bardziej wymagający czasowo	czujniki temperatury DS18B20

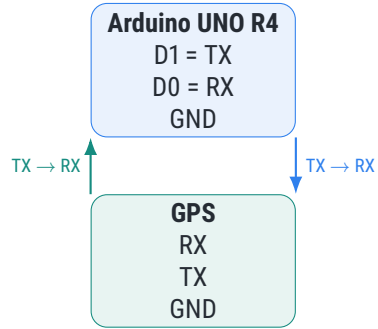
W tym wykładzie: UART + I^2C. SPI i 1-Wire rozwiniemy na Wykładzie 2.

UART — ramka, baud rate i połączenie TX-RX



Co oznacza baud rate?

Baud rate to liczba symboli na sekundę. W typowym UART jeden symbol odpowiada jednemu bitowi, więc **9600 baud** $\approx$ **9600 bit/s**. Jeden bit trwa wtedy około **104** μ**s**.



- TX jednej strony łączymy z RX drugiej,
- potrzebna jest wspólna masa,
- oba urządzenia ustawiamy na ten sam baud rate.

Serial i Serial1 — dwie różne role

Serial — komunikacja z komputerem przez USB

Najwygodniejsze miejsce do diagnostyki: komunikaty trafiają do Serial Monitor.

```
Serial.begin(115200);
Serial.println("Start programu");
```

Serial1 — sprzętowy UART na D0/RX i D1/TX

Tego portu używamy do urządzenia zewnętrznego, np. modułu GPS.

```
Serial1.begin(9600);
Serial1.println("test");
```

Na laboratorium GPS podłączamy do `Serial1`, a `Serial` wykorzystujemy do podglądu odebranych danych na komputerze.

Mostek diagnostyczny: pokaż na komputerze to, co wysyła GPS

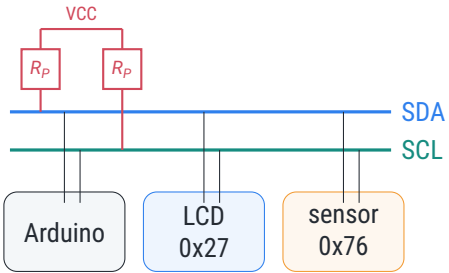
```
void setup() {
  Serial.begin(115200);   // komputer
  Serial1.begin(9600);    // GPS
}

void loop() {
  while (Serial1.available() > 0) {
    char c = Serial1.read();
    Serial.write(c);
  }
}
```

Dlaczego while(available())?

Czytamy tylko te bajty, które już dotarły. Nie zatrzymujemy programu w oczekiwaniu na kolejny znak.

I²C — wiele urządzeń na dwóch przewodach



- SDA = dane, SCL = zegar,
- urządzenia rozróżnia **adres**,
- na Arduino UNO R4 WiFi: SDA=A4, SCL=A5.

Po co rezystory podciągające?

- linie są typu *open-drain* — układy aktywnie wymuszają LOW, ale nie HIGH,
- pull-up przywraca HIGH, gdy nikt nie ściąga linii do GND,
- potrzebujemy **jednego zestawu pull-upów na magistralę**,
- typowo spotyka się kilka kΩ, np. 4.7 kΩ; dobór zależy od pojemności i szybkości,
- wiele modułów ma pull-upy już na płytce.

Źródło: docs.arduino.cc/tutorials/uno-r4-wifi/cheat-sheet/

Pierwszy kontakt z I²C — skaner adresów

```arduino
#include <Wire.h>

void setup() {
  Serial.begin(115200);
  Wire.begin();

  for (byte addr = 1; addr < 127; addr++) {
    Wire.beginTransmission(addr);
    byte err = Wire.endTransmission();

    if (err == 0)
      Serial.println(addr, HEX);
  }
}

void loop() {}
```

Co robią te dwie funkcje?

- `Wire.beginTransmission(addr)` — wybiera urządzenie o podanym **7-bitowym adresie**. Kolejne `Wire.write()` dodawałoby bajty do wysłania.
- `Wire.endTransmission()` — wysyła przygotowaną transmisję i ją kończy. Kod **0 oznacza ACK**: urządzenie odpowiedziało.

Po co skaner?

Nie zapisuje rejestrów. Sprawdza tylko, **które adresy odpowiadają** — to szybki pierwszy krok diagnostyczny.

Schemat komunikacji przez I^2C

```
#include <Wire.h>

void setup() {
  Wire.begin();
}

void loop()
{
  Wire.beginTransmission( adres czujnika );
  Wire.write( adres rejestru ); // np. adres rejestru gdzie
  przechowywana jest wartosc temperatury
  Wire.endTransmission ();
  Wire.requestFrom ( adres czujnika , ile bajtow ma odczytac);
  while(Wire.available())
  {
    unsigned int zmienna = Wire.read();
    ...
  }
}

```

Przykład I²C: odczyt temperatury z LM75

```
#include <Wire.h>
#define addressTemp 0x48

void setup() {
  Wire.begin();
}

void loop()
{
  Wire.beginTransmission(addressTemp);
  Wire.write(0x00); //Patrz: dokumentacja do LM75 (https://labe.engined.eu/data/_uploaded/file/psw/laboratorium/Lab4/
      STM_LM75.pdf) str. 5, tabela nr 2, pierwszy wiersz
  Wire.endTransmission();
  Wire.requestFrom(addressTemp, 2);
  while(Wire.available()) {
    msb = Wire.read();
    lsb = Wire.read();
    lsb = (lsb & 0x80 ) >> 7; //Pobranie D7, patrz: str 6, tab. nr 3,
    temp = msb + 0.5 * lsb;
    Serial.print(" T=");
    Serial.print(temp);
  }
}
```

Liczby szesnastkowe — skąd w kodzie `0x48` i `0x80`?

- przejście z zapisu binarnego na dziesiętny: $0101 \rightarrow 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 4 + 1 = 5$
- inny przykład:
  $0100\ 1000 \rightarrow 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 64 + 8 = 72,$
- przedrostek `0x` oznacza zapis **szesnastkowy** (hexadecimal),
- używamy cyfr 0–9 oraz liter A–F; np. A = 10, F = 15,
- jedna cyfra hex odpowiada dokładnie **4 bitom**, dlatego taki zapis jest wygodny przy rejestrach i maskach bitowych.

Zapis hex	Zapis binarny	Dziesiętnie	W naszym przykładzie
`0x00`	`0000 0000`	0	adres rejestru temperatury LM75
`0x48`	`0100 1000`	72	adres I^2C czujnika LM75
`0x80`	`1000 0000`	128	maska wybierająca najstarszy bit D7

Najważniejsze

To są te same liczby zapisane w inny sposób. Mikrokontroler przechowuje bity; zapis hex ułatwia nam ich czytanie.

Operacje bitowe: `&` wybiera bity, a `|` je ustawia

AND bitowe: `&` — maskowanie

Operacja jest wykonywana **osobno dla każdej pary bitów**. Wynik ma 1 tylko tam, gdzie oba bity są równe 1.

```
  1010 0110     przykładowy bajt lsb
& 1000 0000     maska 0x80
-------------
  1000 0000     pozostaje tylko bit D7
```

OR bitowe: `|` — ustawianie wybranych bitów

```
  0010 0110     przykładowy bajt lsb
| 1000 0000     maska 0x80
-------------
  1010 0110     Ustawia ostatni bit D7 na 1
                nie zmieniając pozostałych bitów.
```

Nie mylimy operatorów

`&` i `|` działają na **bitach**, natomiast `&&` i `||` są operatorami **logicznymi** używanymi np. w warunkach `if`.

Przesunięcia bitowe » i «

- `»` przesuwa bity w prawo — używamy go m.in. do **wydobywania pola bitowego**,
- `«` przesuwa bity w lewo — przydaje się np. do **tworzenia masek** lub składania danych z kilku pól.

Przykład

```
1000 0000 » 1 = 0100 0000
1111 1111 » 7 = 0000 0001
0000 0001 « 3 = 0000 1000
```

Jak czytać linię z przykładu LM75?

```
lsb = (lsb & 0x80) >> 7;
```

1. `lsb & 0x80` — zerujemy wszystkie bity poza D7,
2. `» 7` — przesuwamy D7 na pozycję D0,
3. wynik to już tylko **0 albo 1**, więc można użyć go w `temp = msb + 0.5 * lsb;`.

Dziękuję za uwagę!

Do zobaczenia na laboratorium.