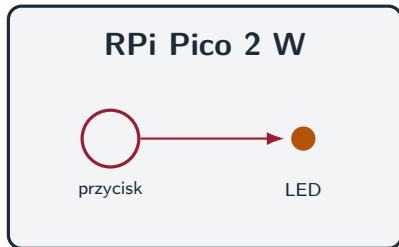


# GPIO, przerwania i timery

Od pojedynczego pinu do reakcji na zdarzenia

Podstawy Systemów Mikroprocesorowych ■ Wykład 2  
dr inż. Dariusz Tefelski

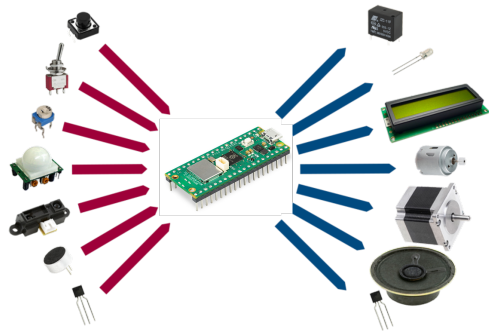
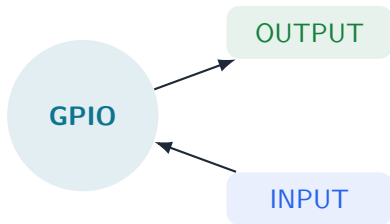




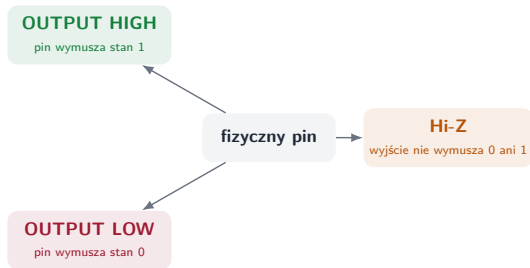
- ▶ Jak odczytać przycisk?
- ▶ Jak wykryć **moment** naciśnięcia?
- ▶ Jak nie zatrzymywać całego programu?
- ▶ Jak wykonywać zadanie okresowo?

# GPIO — General Purpose Input/Output

- ▶ GPIO to cyfrowe wejścia/wyjścia ogólnego przeznaczenia.
- ▶ Ten sam fizyczny pin może pełnić różne funkcje zależnie od konfiguracji.
- ▶ W trybie GPIO program może **ustawiać** albo **odczytywać** poziom logiczny.



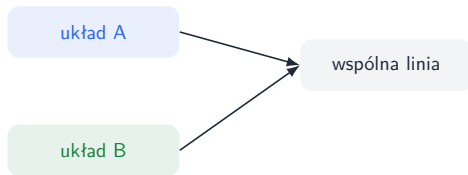
# Fizyczny pin i trzy podstawowe stany pracy



- ▶ **Hi-Z** (*high impedance*) oznacza, że stopień wyjściowy pinu jest wyłączony.
- ▶ Pin praktycznie **nie narzuca napięcia** na linii – może je ustalić zewnętrzny układ.
- ▶ Typowy tryb INPUT bez pull-up/pull-down jest z punktu widzenia wyjścia stanem o dużej impedancji.
- ▶ Hi-Z jest potrzebny m.in. podczas **odczytu wejścia**, współdzielenia linii przez kilka układów i w interfejsach dwukierunkowych.

**Ważne:** Hi-Z nie oznacza „napięcie 0 V”. Oznacza: **pin nie steruje linią.**

# Po co właściwie potrzebujemy Hi-Z?

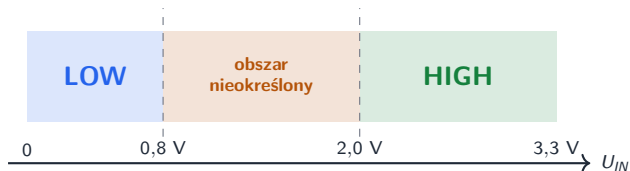


tylko jeden układ powinien aktywnie sterować linią

- ▶ Gdy układ ma **nadawać** – ustawia LOW albo HIGH.
- ▶ Gdy ma **słuchać** – wyłącza swój driver i przechodzi w Hi-Z.
- ▶ Dzięki temu drugi układ może ustawić napięcie bez „walki” dwóch wyjść.
- ▶ Tę ideę spotkamy później m.in. w liniach dwukierunkowych i magistralach.

Dwa wyjścia ustawione przeciwnie na jednej linii mogą spowodować **duży prąd i błąd sprzętowy**.

# Poziomy logiczne GPIO w RP2350



Dla standardowego wejścia RP2350 przy  $IOVDD = 3,3\text{ V}$ :

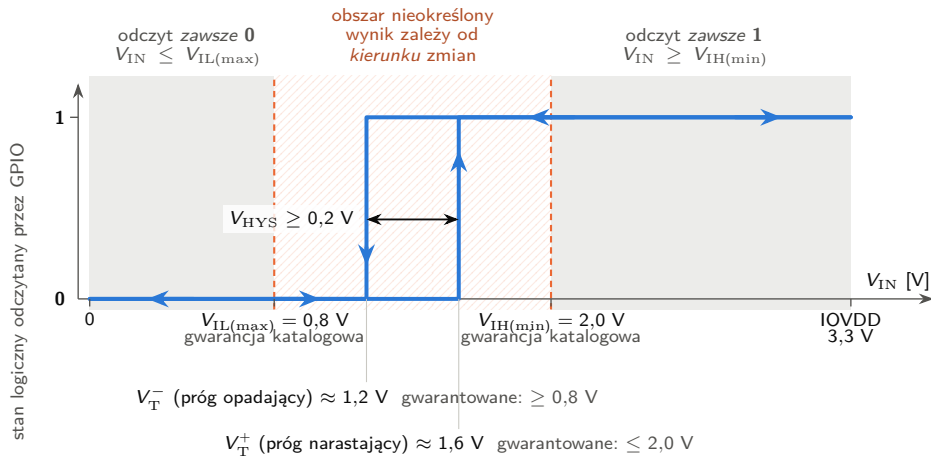
$$V_{IL} \leq 0,8\text{ V}$$

$$V_{IH} \geq 2,0\text{ V}.$$

- ▶ **0–0,8 V**: stan LOW jest gwarantowany.
- ▶ **2,0–3,3 V**: stan HIGH jest gwarantowany.
- ▶ Między 0,8 a 2,0 V nie projektujemy poprawnej logiki cyfrowej – wynik nie jest gwarantowany.
- ▶ Wyjście cyfrowe dąży do okolic **0 V** albo **3,3 V**, ale napięcie zależy też od obciążenia.

Na laboratorium projektujemy sygnały GPIO jako **logikę 3,3 V**. Nie zakładamy, że 5 V jest bezpieczne bez sprawdzenia konkretnego pinu i dokumentacji.

# Schmitt trigger: dlaczego wejście nie przełącza się przy jednym idealnym progu?



Pętla histerezy – strzałki wskazują kierunek zmian  $V_{IN}$ . Źródło: RP2350 Datasheet, rozdz. 14.9.4 *IO electrical characteristics*, Tab. 1683.

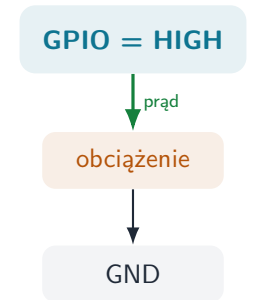
Wartości 1,6 V / 1,2 V są przykładowe (datasheet podaje tylko granice gwarantowane).

- ▶ Wejścia GPIO RP2350 mają możliwość użycia **przerzutnika Schmitta**.
- ▶ W Micropython przerzutnik Schmitta jest domyślnie włączony i API nie daje możliwości wyłączenia, jednakże można dezaktywować go poprzez bezpośredni dostęp do rejestrów konfiguracyjnych.
- ▶ W C API (Pico SDK) dostępne są funkcje sterujące np. `gpio_set_input_hysteresis_enabled()`
- ▶ Próg przełączenia przy rosnącym napięciu i przy malejącym napięciu nie jest identyczny.
- ▶ Różnica progów to **histereza**; przy  $IOVDD = 3,3\text{ V}$  specyfikowana jest na około **0,2 V**.
- ▶ Dzięki temu niewielki szum przy progu nie powoduje szybkiego przełączania 0/1/0/1.
- ▶ Dlaczego jest opcja wyłączenia przerzutnika Schmitta, chociaż ma tak dużo zalet? Ponieważ może wprowadzać niewielkie opóźnienie, które może być istotne przy odbiorze sygnałów z bardzo szybkich interfejsów cyfrowych.



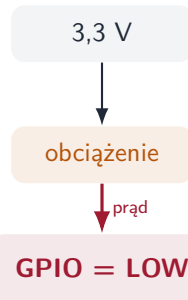
Program wybiera stan **0** lub **1**, a układ wyjściowy wymusza odpowiedni poziom na pinie.

# Jak płynie prąd przez wyjście GPIO?



**GPIO dostarcza prąd**

ang. *source current*

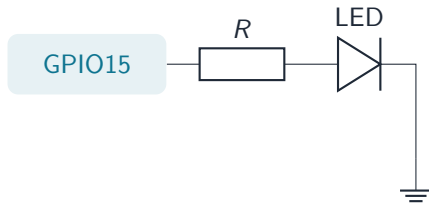


**GPIO odprowadza prąd do GND**

ang. *sink current*

W obu przypadkach prąd musi być ograniczony przez obciążenie lub rezystor.

# LED wymaga ograniczenia prądu



$$R = \frac{V_{GPIO} - V_F}{I_{LED}}$$

Dla czerwonej LED przyjmijmy:

$$V_{GPIO} \approx 3,3 \text{ V}, \quad V_F \approx 1,7 \text{ V}.$$

Dla prądu około 4 mA:

$$R = \frac{3,3 - 1,7}{0,004} = 400 \Omega.$$

Najbliższa typowa wartość: 390  $\Omega$ .

| Rezystor     | Prąd (w przybliżeniu) |
|--------------|-----------------------|
| 220 $\Omega$ | 7,3 mA                |
| 330 $\Omega$ | 4,8 mA                |
| 390 $\Omega$ | 4,1 mA                |
| 470 $\Omega$ | 3,4 mA                |

Są to wartości orientacyjne –  $V_F$  zależy od konkretnej diody, a napięcie GPIO zmienia się z obciążeniem.

### GPIO

2 mA

4 mA

8 mA

12 mA

- ▶ RP2350 pozwala ustawić **siłę sterowania wyjścia** (*drive strength*).
- ▶ Dostępne poziomy to **2, 4, 8 i 12 mA**.
- ▶ Wartość określa warunki, dla których producent gwarantuje poziomy  $V_{OH}$  i  $V_{OL}$ .
- ▶ To **nie jest ogranicznik prądu**: pin nie „ustawia automatycznie” zadanej wartości prądu.
- ▶ W standardowym API `machine.Pin` portu RP2 nie ma obecnie argumentu `drive=` — ustawienie wymaga niższego poziomu albo Pico SDK.

Dla początkującego najważniejsza zasada: **GPIO steruje sygnałem, nie zasila dużych odbiorników.**

# Jak ustawić drive strength?

## MicroPython RP2 — bezpośrednio przez rejestr pada:

```
from machine import mem32

PADS_BANK0 = 0x40038000
gpio = 15
reg = PADS_BANK0 + 0x04 + 4 * gpio

DRIVE_MASK = 0b11 << 4
DRIVE_8MA = 0b10 << 4

mem32[reg] = (mem32[reg] & ~DRIVE_MASK) | DRIVE_8MA
```

Bity DRIVE[5:4]:

| Kod | Siła  |
|-----|-------|
| 00  | 2 mA  |
| 01  | 4 mA  |
| 10  | 8 mA  |
| 11  | 12 mA |

## Pico SDK (C/C++):

```
gpio_set_drive_strength(
    15,
    GPIO_DRIVE_STRENGTH_8MA
);
```

Bezpośrednia zmiana rejestru jest rozwiązaniem zaawansowanym — trzeba zachować pozostałe bity konfiguracji pada, dlatego używamy maski.

```
from machine import Pin

led = Pin(15, Pin.OUT)

led.value(1) # HIGH
led.value(0) # LOW

# wygodne odpowiedniki:
led.on()
led.off()

# zmiana na stan przeciwny:
led.toggle()
```

Pin(15, Pin.OUT)



ustawianie 0 / 1



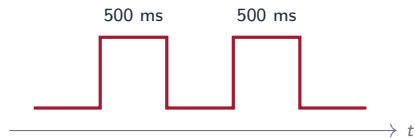
toggle(): 0↔1

## Przykład: mrugająca dioda

```
from machine import Pin
from time import sleep_ms

led = Pin(15, Pin.OUT)

while True:
    led.toggle()
    sleep_ms(500)
```



Każde wywołanie `toggle()` zmienia stan diody na przeciwny.

# GPIO jako wejście



Program **nie ustawia** napięcia. **Sprawdza, jaki stan jest obecnie na pinie.**

## Przycisk bez pull-up/pull-down: wejście pływające



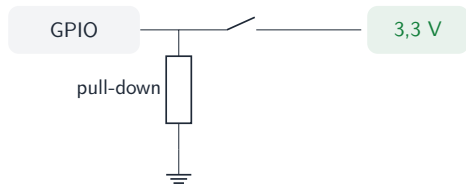
Gdy przycisk jest otwarty:

- ▶ zewnętrzny układ **nie ustala napięcia** na GPIO,
- ▶ wejście ma dużą impedancję, więc pobiera bardzo mały prąd,
- ▶ pojemności pasożytnicze i zakłócenia mogą przesuwac napięcie,
- ▶ odczyt może zmieniać się między 0 i 1.

Takie wejście nazywamy **floating input**.  
Potrzebujemy **stanu domyślnego**: pull-up albo pull-down.

# Pull-down i pull-up: ustawiamy stan domyślny

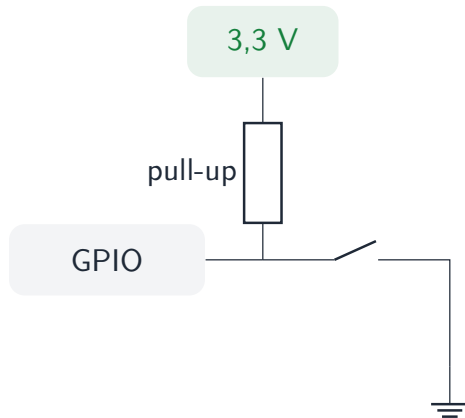
## Pull-down: domyślnie 0



**Przycisk otwarty:**  $\text{GPIO} \approx 0 \text{ V} \Rightarrow 0$

**Przycisk zamknięty:**  $\text{GPIO} \approx 3,3 \text{ V} \Rightarrow 1$

## Pull-up: domyślnie 1

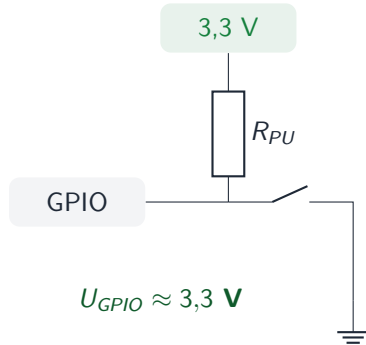


**Przycisk otwarty:**  $\text{GPIO} \approx 3,3 \text{ V} \Rightarrow 1$

**Przycisk zamknięty:**  $\text{GPIO} \approx 0 \text{ V} \Rightarrow 0$

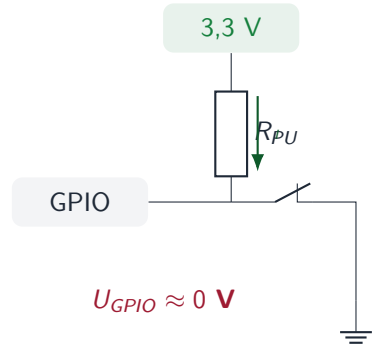
# Dlaczego przy pull-up naciśnięty przycisk daje 0?

## Przycisk otwarty



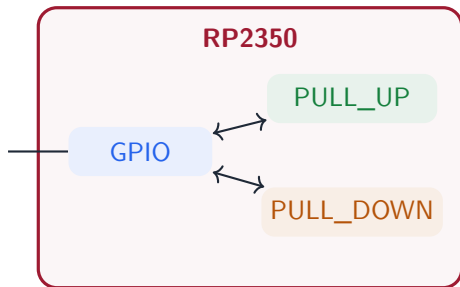
Wejście ma bardzo dużą impedancję, więc prawie nie pobiera prądu. Rezystor podciąga GPIO do 3,3 V  $\Rightarrow$  **odczyt 1.**

## Przycisk naciśnięty



Zwarcie przez przycisk ustala GPIO blisko 0 V. Prąd płynie z 3,3 V przez  $R_{PU}$  do GND  $\Rightarrow$  **odczyt 0.**

# W praktyce nie potrzebujemy zewnętrznego rezystora



Wewnętrzne rezystory wystarczają zwykle do ustalenia stanu typowego przycisku.

Przy  $IOVDD = 3,3\text{ V}$  ich wartości nie są precyzyjne:

$$R_{PU} = 32 \dots 86\text{ k}\Omega$$

$$R_{PD} = 36 \dots 113\text{ k}\Omega$$

- ▶ włączamy je programowo,
- ▶ są dobre do ustawienia **domyślnego stanu wejścia**,
- ▶ nie traktujemy ich jak dokładnych rezystorów pomiarowych.

```
from machine import Pin  
  
button = Pin(14, Pin.IN, Pin.PULL_UP)  
  
state = button.value()  
print(state)
```

**1** przycisk puszczony

`button.value()` → **1**

**0** przycisk naciśnięty

`button.value()` → **0**

# Czas na podsumowanie tej części wykładu!

GPIO ▪ pull-up ▪ pull-down

Co może się stać, gdy cyfrowe wejście GPIO pozostawimy bez pull-up, pull-down i bez aktywnego źródła sygnału?

- A. Zawsze odczytamy 0
- B. Odczyt może być niestabilny i podatny na zakłócenia
- C. Pin automatycznie przełączy się w tryb wyjściowy
- D. Mikrokontroler zresetuje się

Co może się stać, gdy cyfrowe wejście GPIO pozostawimy bez pull-up, pull-down i bez aktywnego źródła sygnału?

A. Zawsze odczytamy 0

B. Odczyt może być niestabilny i podatny na zakłócenia

C. Pin automatycznie przełączy się w tryb wyjściowy

D. Mikrokontroler zresetuje się

Przycisk jest połączony między GPIO a GND, a wejście skonfigurowano jako `Pin.PULL_UP`. Co odczytamy po naciśnięciu przycisku?

A. 0

B. 1

C. 3,3

D. losowo 0/1

Przycisk jest połączony między GPIO a GND, a wejście skonfigurowano jako `Pin.PULL_UP`. Co odczytamy po naciśnięciu przycisku?

**A. 0**

B. 1

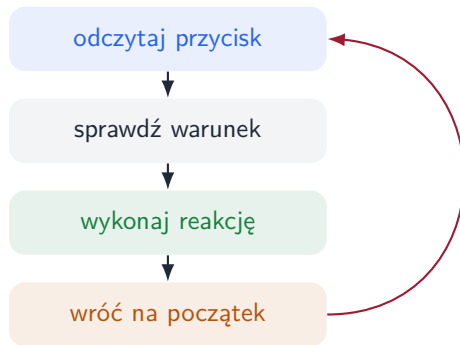
C. 3,3

D. losowo 0/1

# Polling i wykrywanie zmian stanu

Jak program zauważy, że coś się wydarzyło?

## Polling — najprostszy sposób reagowania



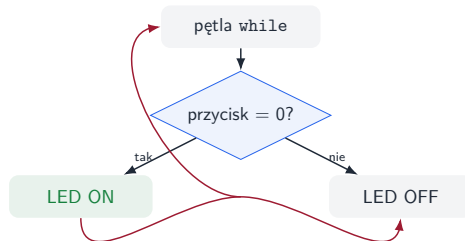
**Polling** = program regularnie pyta urządzenie o jego stan.

# Polling: kod i schemat blokowy

```
from machine import Pin

button = Pin(14, Pin.IN, Pin.PULL_UP)
led = Pin(15, Pin.OUT)

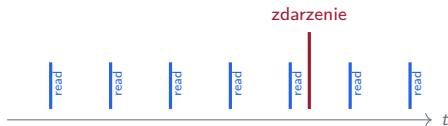
while True:
    if button.value() == 0:
        led.on()
    else:
        led.off()
```



To jest poprawne i często wystarczające rozwiązanie: program regularnie **sprawdza stan wejścia**.

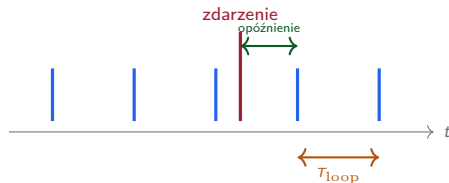
# Polling: jak często sprawdzamy wejście i jak szybko reagujemy?

## CPU regularnie wykonuje odczyt



Odczyty odbywają się także wtedy, gdy nic się nie dzieje.

## Reakcja nastąpi przy następnym odczycie



W najprostszym przypadku:

$$t_{reakcji,max} \approx T_{loop}.$$

## Zalety

- ▶ prosty do napisania i debugowania,
- ▶ kolejność działania jest czytelna,
- ▶ dobry przy małej liczbie wejść,
- ▶ często wystarcza w małych systemach.

## Ograniczenia

- ▶ CPU regularnie pyta o stan,
- ▶ reakcja zależy od częstotliwości sprawdzania,
- ▶ zbyt długa pętla może „przegapić” krótkie zdarzenie,
- ▶ trudniej skalować do wielu zadań.

## Problem zaczyna się, gdy program robi kilka rzeczy

sprawdź przycisk

odczytaj czujnik

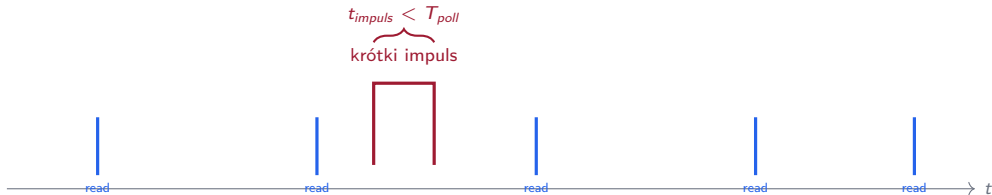
wyślij dane

steruj silnikiem

odśwież ekran

A co, jeśli ważne zdarzenie pojawi się **pomiędzy** kolejnymi odczytami?

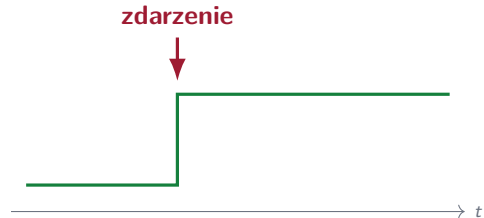
## Krótki impuls może pojawić się między dwoma odczytami



Jeśli sygnał wróci do poprzedniego stanu przed następnym odczytem, program może **nigdy go nie zobaczyć**.

To jeden z powodów, dla których krótkie i ważne zdarzenia często obsługuje się przerwaniem albo sprzętowymi peryferiami.

# Poziom czy zmiana stanu?



Często interesuje nas nie stan „jest naciśnięty”, lecz moment: **„właśnie został naciśnięty”**.

## Zbocze można wykryć także bez przerwania

```
previous = button.value()

while True:
    current = button.value()

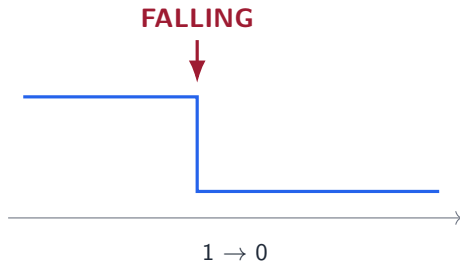
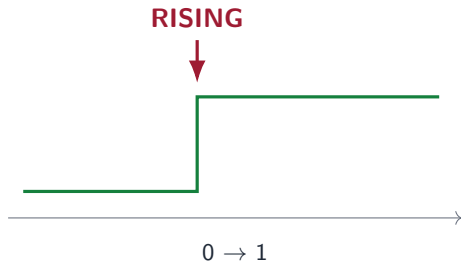
    if previous == 1 and current == 0:
        print("Nacisnieto")

    previous = current
```

- ▶ Polling może wykrywać nie tylko **poziom**, ale też **zmianę poziomą**.
- ▶ Porównujemy stan bieżący ze stanem poprzednim.
- ▶ Dla pull-up przejście  $1 \rightarrow 0$  oznacza naciśnięcie.

To nadal polling – program musi wykonać odczyt wystarczająco często.

# Zbocze narastające i opadające



Przy przycisku z pull-up naciśnięcie najczęściej odpowiada **zboczu opadającemu**.

# Rzeczywisty przycisk: styki drgają

## Na schemacie



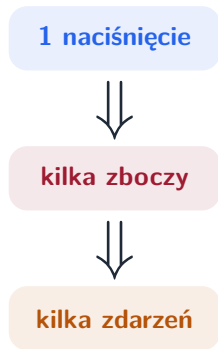
Wygląda jak jedno idealne przełączenie.

## W rzeczywistości



Mechaniczna sprężystość styków powoduje **bouncing**.

# Jedno naciśnięcie może dać wiele zboczy — potrzebujemy debouncingu

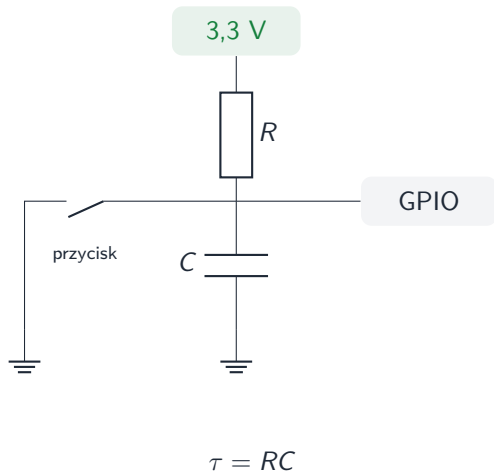


Drgania styków możemy ograniczyć na dwa podstawowe sposoby:

- ▶ **sprzętowo** — filtrujemy sygnał elektryczny, np. układem RC, a następnie wykorzystujemy wejście z histerezą Schmitta;
- ▶ **programowo** — po wykryciu zdarzenia przez krótki czas ignorujemy kolejne zmiany albo śledzimy czas i stan przycisku.

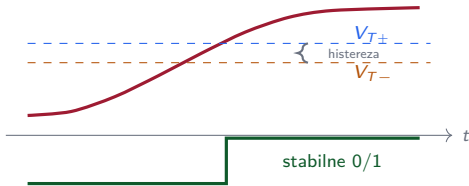
W obu przypadkach cel jest ten sam: **jedno logiczne zdarzenie** dla jednego fizycznego naciśnięcia.

## Sprzętowy debounce: filtr RC wygładza szybkie zmiany



- ▶ Kondensator nie pozwala napięciu zmieniać się natychmiast.
- ▶ Krótkie impulsy od drgań styków są tłumione.
- ▶ Przykład:  $R = 10\text{ k}\Omega$  i  $C = 1\text{ }\mu\text{F}$  daje  $\tau = 10\text{ ms}$ .
- ▶ Po naciśnięciu kondensator rozładowuje się przez przycisk; po puszczeniu ładuje się przez  $R$ .
- ▶ Wartości  $R$  i  $C$  dobieramy do czasu drgań i wymaganej szybkości reakcji.

# Sprzętowy debounce: po filtrze RC wykorzystujemy wejście Schmitta

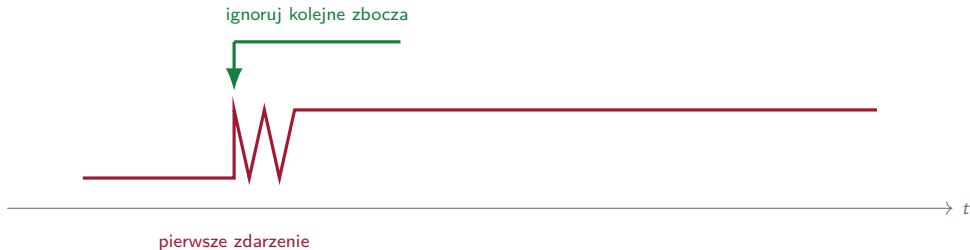


RC wygładza drgania, ale napięcie przechodzi przez zakres pośredni. **Schmitt trigger** używa dwóch różnych progów przy narastaniu i opadaniu napięcia, dzięki czemu mały szum nie powoduje serii przełączeń.

## Jak korzystamy z tego na RPi Pico 2 W?

- ▶ wyjście filtru RC podłączamy bezpośrednio do wejścia GPIO;
- ▶ w RP2350 wejścia GPIO mają funkcję Schmitta, która jest **domyślnie włączona** w rejestrach padów;
- ▶ w zwykłym MicroPythonie nie podajemy osobnego argumentu typu `schmitt=True`; konfigurujemy wejście normalnie, np. `Pin(14, Pin.IN, Pin.PULL_UP)`;
- ▶ jawna zmiana ustawienia Schmitta wymaga niższego poziomu: rejestrów padów lub Pico SDK.

## Debounce programowy: krótki „okres ochronny”



Po zaakceptowaniu zdarzenia czekamy, aż styki się uspokoją.

## Debounce bez blokowania: pamiętamy czas ostatniego zdarzenia

```
from time import ticks_ms, ticks_diff

last_press = -1000

while True:
    if button.value() == 0:
        now = ticks_ms()

        if ticks_diff(now, last_press) > 30:
            last_press = now
            print("Nacisnieto")

    # program moze wykonywac inne zadania
```

Nie zatrzymujemy pętli przez `sleep_ms(30)`. Kolejne odczyty nadal się odbywają, ale zdarzenia w krótkim oknie czasu są ignorowane.

# Prosty debounce w pętli

```
from machine import Pin
from time import sleep_ms

button = Pin(14, Pin.IN, Pin.PULL_UP)

while True:
    if button.value() == 0:
        print("Nacisnieto")
        sleep_ms(30) # bardzo prosty debounce
        while button.value() == 0:
            pass # czekamy na puszczenie
```

To rozwiązanie jest czytelne, ale **blokuje główną pętlę**. Później zrobimy to lepiej.

# Czas na podsumowanie tej części wykładu!

polling ▀ zbocza ▀ bouncing

Które zdanie o pollingu jest najbardziej poprawne?

A. Polling jest zawsze błędem projektowym

B. Polling jest prosty i bywa wystarczający, ale wymaga regularnego sprawdzania wejścia

C. Polling działa wyłącznie z timerem

D. Polling automatycznie eliminuje drgania styków

Które zdanie o pollingu jest najbardziej poprawne?

A. Polling jest zawsze błędem projektowym

**B. Polling jest prosty i bywa wystarczający, ale wymaga regularnego sprawdzania wejścia**

C. Polling działa wyłącznie z timerem

D. Polling automatycznie eliminuje drgania styków

Dlaczego jedno naciśnięcie mechanicznego przycisku może wygenerować kilka wykrytych zboczy?

A. przez drgania styków

B. przez pamięć Flash

C. przez dwa rdzenie CPU

D. przez zmianę częstotliwości zegara

Dlaczego jedno naciśnięcie mechanicznego przycisku może wygenerować kilka wykrytych zboczy?

A. przez drgania styków

B. przez pamięć Flash

C. przez dwa rdzenie CPU

D. przez zmianę częstotliwości zegara

# Przerwania GPIO

Zdarzenie może samo zgłosić się do procesora

## Polling

program co chwilę sprawdza:  
„czy zdarzenie już wystąpiło?”

Dobra analogia: co chwilę podchodzę do drzwi i sprawdzam, czy ktoś stoi.

## Przerwanie (IRQ)

program wyko-  
nuje inne zadania,  
a zdarzenie zgła-  
sza potrzebę obsługi

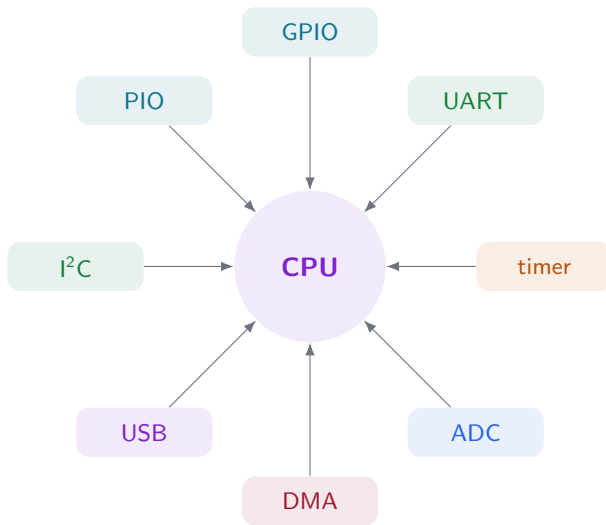
Dobra analogia: pracuję, a gość naciska dzwonek do drzwi.

Przerwanie nie jest „lepszym pollingiem”. To inny mechanizm reakcji na zdarzenie.

## Co dzieje się po zgłoszeniu przerwania?

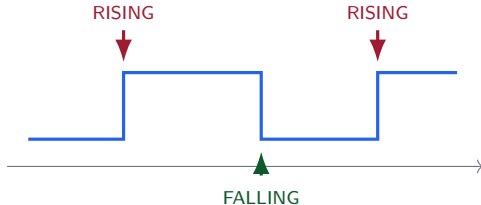


## Skąd mogą pochodzić przerwania?



Dzisiaj skupiamy się na **GPIO** i zdarzeniach czasowych.

# Przerwanie GPIO: jakie zdarzenie ma być wyzwalaczem?



- ▶ W porcie **RP2 MicroPythona** klasa `Pin` udostępnia dla GPIO:
  - `IRQ_RISING`,
  - `IRQ_FALLING`.
- ▶ Można je łączyć operatorem `|`.

Sam sprzęt RP2350 potrafi generować IRQ także od utrzymującego się poziomu HIGH/LOW. Pico SDK udostępnia `GPIO_IRQ_LEVEL_HIGH/LOW`, ale port RP2 MicroPythona nie wystawia tych stałych przez `machine.Pin`.

## Trigger dobieramy do sposobu podłączenia przycisku

| Konfiguracja                 | Stan spoczynkowy | Naciśnięcie       | Naturalny trigger |
|------------------------------|------------------|-------------------|-------------------|
| pull-up, przycisk do GND     | HIGH (1)         | $1 \rightarrow 0$ | IRQ_FALLING       |
| pull-down, przycisk do 3,3 V | LOW (0)          | $0 \rightarrow 1$ | IRQ_RISING        |

Nie wybieramy RISING/FALLING „na pamięć”. Najpierw pytamy: **jaki poziom jest przed naciśnięciem i po nim?**

## Przykład: przerwanie po naciśnięciu przycisku

```
from machine import Pin

button = Pin(14, Pin.IN, Pin.PULL_UP)

def on_button(pin):
    print("Nacisnieto")

button.irq(
    trigger=Pin.IRQ_FALLING,
    handler=on_button
)

while True:
    pass # tutaj moze dzialac program glowny
```

Pętla główna nie musi sprawdzać przycisku. Może wykonywać inne zadania, a callback zostanie wywołany po zdarzeniu.

## Przykład: reakcja na naciśnięcie i puszczenie przycisku

```
from machine import Pin

button = Pin(14, Pin.IN, Pin.PULL_UP)

def on_change(pin):
    if pin.value() == 0:
        print("Nacisnieto")
    else:
        print("Puszczono")

button.irq(
    trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING,
    handler=on_change
)

while True:
    pass # tutaj moze dzialac program glowny
```

Przy pull-up: FALLING = naciśnięcie, RISING = puszczenie. Operator | łączy oba wyzwalacze; bouncing nadal może wygenerować kilka IRQ.

## Skąd w handlerze bierze się argument `pin`?



MicroPython przekazuje do funkcji obiekt `Pin`, który wywołał zdarzenie.

# Soft IRQ i hard IRQ: gdzie ustawiamy tryb?

**hard=False**

zdarzenie sprzętowe



MicroPython planuje callback



callback wykonany później

Domyślne ustawienie `Pin.irq()`. Większa wygoda kodu Pythona, ale możliwa większa latencja i jitter.

**hard=True**

zdarzenie sprzętowe



twardy kontekst IRQ



callback bezpośrednio

Mniejsza latencja i jitter, ale handler podlega ograniczeniom twardego przerwania.

```
button.irq(handler=on_button,  
            trigger=Pin.IRQ_FALLING,  
            hard=False) # zmien na hard=True dla hard IRQ
```

# Hard IRQ: dlaczego kod musi być szczególnie prosty?

## bezpieczny wzorzec

- ▶ ustaw prostą flagę,
- ▶ zwiększ prosty licznik,
- ▶ zapisz prostą wartość,
- ▶ szybko zakończ handler.

## w hard IRQ unikaj

- ▶ tworzenia nowych obiektów,
- ▶ rozbudowanych napisów i formatowania,
- ▶ operacji wymagających alokacji pamięci,
- ▶ długich obliczeń i oczekiwania.

W `hard=True` MicroPython blokuje GC i alokację pamięci na czas callbacka. Na tym kursie, o ile nie potrzebujemy bardzo małej latencji, pozostajemy przy `hard=False`.

# Handler przerwania powinien być krótki

## Nie

długie obliczenia  
`sleep()`  
dużo komunikacji  
rozbudowane `print()`

## Lepiej

ustaw flagę  
zapisz prostą informację  
szybko wróć

## Wzorzec: przerwanie tylko zgłasza zdarzenie

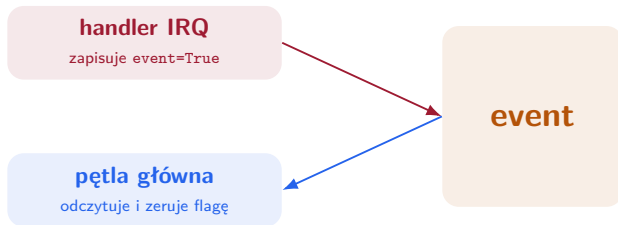
```
event = False

def on_button(pin):
    global event
    event = True

while True:
    if event:
        event = False
        # tutaj wykonujemy dluzsza reakcje
        print("Obsługa przycisku")
```

Handler jest krótki, a właściwa praca odbywa się w głównej części programu.

# Handler i pętla główna korzystają ze wspólnych danych



- ▶ Prosta flaga jest wygodnym sposobem przekazania informacji.
- ▶ Im bardziej złożone wspólne dane, tym większe ryzyko, że kod główny i callback „spotkają się” w złym momencie.
- ▶ Dlatego handler powinien wykonywać **minimum pracy** i zostawiać logikę aplikacji pętli głównej.

## Sekcja krytyczna: gdy kilka operacji musi być niepodzielnych

```
import machine

state = machine.disable_irq()

# fragment, ktorego IRQ nie powinno
# przerwac w polowie:
a = shared_a
b = shared_b
shared_a = 0
shared_b = 0

machine.enable_irq(state)
```

- ▶ Wyłączamy IRQ tylko na **bardzo krótki** czas.
- ▶ Zachowujemy poprzedni stan zwrócony przez `disable_irq()`.
- ▶ Ten sam stan przekazujemy później do `enable_irq(state)`.
- ▶ Nie stosujemy tego do długich obliczeń ani oczekiwania.

Sekcja krytyczna chroni **spójność wspólnych danych**. Nie jest zamiennikiem dobrze zaprojektowanego handlera.

## Flaga czy licznik zdarzeń? To nie zawsze to samo

### flaga

`event = True`

mówi: „co najmniej jedno  
zdarzenie wystąpiło”

### licznik

`count += 1`

pozwała rozróż-  
nić kilka zdarzeń

Jeśli dwa IRQ pojawią się zanim pętla wyzeruje flagę, sama flaga nadal będzie miała tylko wartość True.  
Dobór sposobu przekazania informacji zależy od tego, czy ważna jest **liczba zdarzeń**.

## Przerwanie i debounce — razem

```
from machine import Pin
from time import ticks_ms, ticks_diff

last = 0

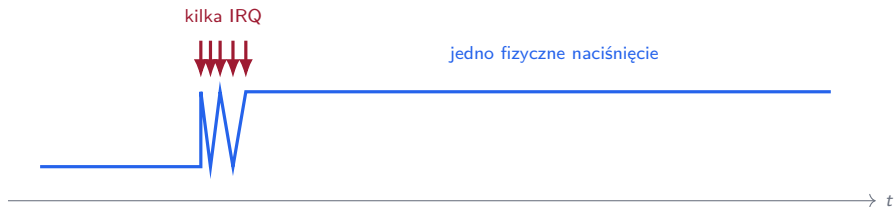
def on_button(pin):
    global last
    now = ticks_ms()
    if ticks_diff(now, last) > 30:
        last = now
        # zaakceptowane naciśnięcie

button = Pin(14, Pin.IN, Pin.PULL_UP)
button.irq(trigger=Pin.IRQ_FALLING,
           handler=on_button)

while True:
    pass # tutaj może działać reszta programu
```

To prosty wzorzec dydaktyczny; w większym systemie jeszcze lepiej przekazać zdarzenie do kodu głównego.

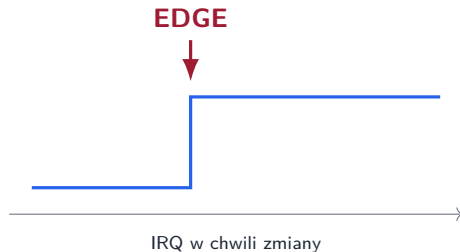
## Przerwanie nie usuwa drgań styków



IRQ reaguje na każde skonfigurowane zbocze – także na zbocza powstałe przez **bouncing**.

Dlatego debounce jest potrzebny zarówno przy pollingu, jak i przy przerwaniach.

## Edge-triggered i level-triggered — to nie to samo



Sprzęt RP2350 obsługuje oba rodzaje zdarzeń GPIO. W **MicroPythonie na porcie RP2** przez `Pin.irq()` używamy jednak `IRQ_RISING` i `IRQ_FALLING`; level-triggered jest dostępne na niższym poziomie, np. w Pico SDK.

# Priorytet i latencja — dwie rzeczy, o których warto wiedzieć

## priorytet

co jest ważniejsze, gdy  
kilka zdarzeń konkuruje?

## latencja

ile czasu mija od  
zdarzenia do reakcji?

## Polling

- ▶ prosty i czytelny,
- ▶ dobry dla wolnych zmian i krótkiej pętli,
- ▶ łatwy do debugowania,
- ▶ naturalny, gdy i tak regularnie wykonujemy wiele odczytów.

## IRQ

- ▶ dobry dla krótkich lub rzadkich zdarzeń,
- ▶ pozwala reagować niezależnie od miejsca w pętli,
- ▶ wymaga ostrożnego projektowania handlera,
- ▶ może komplikować przepływ programu.

Najlepszy system często używa **obu**: polling dla części zadań i IRQ dla zdarzeń wymagających szybkiej reakcji.

# Czas na podsumowanie tej części wykładu!

IRQ ▪ handler ▪ soft/hard

Przycisk z pull-up ma wygenerować przerwanie w chwili naciśnięcia.  
Który trigger jest najbardziej naturalny?

A. `Pin.IRQ_RISING`

B. `Pin.IRQ_FALLING`

C. `Pin.OUT`

D. `Timer.PERIODIC`

Przycisk z pull-up ma wygenerować przerwanie w chwili naciśnięcia.  
Który trigger jest najbardziej naturalny?

A. `Pin.IRQ_RISING`

B. `Pin.IRQ_FALLING`

C. `Pin.OUT`

D. `Timer.PERIODIC`

Który projekt handlera przerwania jest najlepszy?

A. Handler zatrzymuje się na 3 s przez `sleep(3)`, zanim wróci do programu

B. Handler ustawia flagę, a dłuższa obsługa odbywa się w pętli głównej

C. Handler wykonuje pełną transmisję sieciową

D. Handler zawiera nieskończoną pętlę

Który projekt handlera przerwania jest najlepszy?

A. Handler zatrzymuje się na 3 s przez `sleep(3)`, zanim wróci do programu

**B. Handler ustawia flagę, a dłuższa obsługa odbywa się w pętli głównej**

C. Handler wykonuje pełną transmisję sieciową

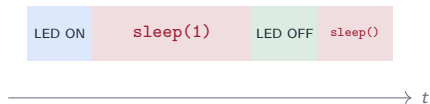
D. Handler zawiera nieskończoną pętlę

# Czas i timery

Jak wykonywać zadania w odpowiednim momencie bez blokowania programu?

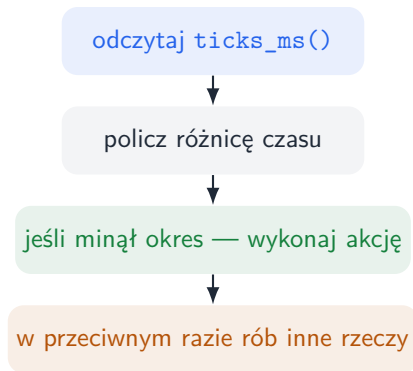
## `sleep()` jest wygodny, ale blokuje główny tok programu

- ▶ `sleep(1)` jest prosty i czytelny dla sekwencji krok po kroku.
- ▶ Główna pętla Pythona nie wykonuje w tym czasie następnych instrukcji.
- ▶ Przerwania mogą nadal zostać obsłużone.
- ▶ Przy wielu zadaniach potrzebujemy podejścia nieblokującego.



Kod znajdujący się za `sleep()` musi poczekać do końca opóźnienia.

## Nieblokująco: sprawdzamy, ile czasu minęło



W MicroPythonie do różnic czasu używamy `ticks_diff()`, bo licznik `ticks` okresowo się zawija.

## `ticks_ms()` się zawija – dlatego używamy `ticks_diff()`



Nie zapisujemy:

`now - last`

jako ogólnej metody dla zegara ticks.

Używamy:

`ticks_diff(now, last)`

- ▶ funkcja uwzględnia cykliczne zawijanie licznika,
- ▶ dzięki temu kod działa poprawnie także po długim czasie pracy.

# Miganie LED bez sleep()

```
from machine import Pin
from time import ticks_ms, ticks_diff

led = Pin(15, Pin.OUT)
last = ticks_ms()

while True:
    now = ticks_ms()
    if ticks_diff(now, last) >= 500:
        last = now
        led.toggle()

    # tutaj mozna wykonywac inne zadania
```

## Timer: „zawołaj mnie później”



Timer pozwala uruchomić reakcję **po czasie** albo **okresowo**.

## Timer: okres i częstotliwość opisują to samo

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

Jeśli  $T$  podajemy w sekundach,  $f$  otrzymujemy w hercach.

| Częstotliwość | Okres  |
|---------------|--------|
| 1 Hz          | 1 s    |
| 2 Hz          | 500 ms |
| 10 Hz         | 100 ms |
| 100 Hz        | 10 ms  |

W `Timer.init()` możemy myśleć „co ile?” (period) albo „ile razy na sekundę?” (freq).

## RP2350

### TIMER0

64-bit counter

4 alarmy

### TIMER1

64-bit counter

4 alarmy

### AON timer

Always-On Timer

niski pobór mocy

- ▶ TIMER0 i TIMER1 tworzą precyzyjną bazę czasu i mogą zgłaszać przerwania po osiągnięciu ustawionego czasu.
- ▶ **AON = Always-On**: osobny timer w domenie zasilania działający także w trybach oszczędzania energii.

## sprzęt RP2350

`TIMER0/TIMER1` mają fizyczne liczniki i po 4 alarmy

## MicroPython RP2

standardowe `machine.Timer`  
nie pozwala wybrać  
bezpośrednio `TIMER0` ani `TIMER1`

Na porcie RP2 klasa `machine.Timer` udostępnia **wyłącznie timery wirtualne**. Zapis `Timer(0)` lub `Timer(1)` nie oznacza więc wyboru sprzętowego `TIMER0/TIMER1`.

# ONE\_SHOT i PERIODIC



```
from machine import Pin, Timer

led = Pin(15, Pin.OUT)
tim = Timer()

def tick(timer):
    led.toggle()

tim.init(period=500,
          mode=Timer.PERIODIC,
          callback=tick,
          hard=False)

while True:
    pass # tutaj moze dzialac reszta programu
```

Callback jest wywoływany co 500 ms, a główny program nie musi wykonywać `sleep(0.5)`.

# MicroPython: przykład timera ONE\_SHOT

```
from machine import Pin, Timer

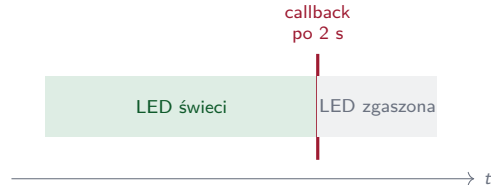
led = Pin(15, Pin.OUT)
tim = Timer()

def turn_off(timer):
    led.off()

led.on() # zapal diode od razu

tim.init(period=2000,
         mode=Timer.ONE_SHOT,
         callback=turn_off,
         hard=False)

while True:
    pass # inne zadania
```



- ▶ ONE\_SHOT wywołuje callback **tylko raz** po zadanym czasie.
- ▶ Po wykonaniu callbacka timer nie powtarza zdarzenia.
- ▶ Kolejne odliczanie uruchamiamy ponownie przez `tim.init(...)`.

# Callback timera ma te same ograniczenia projektowe co handler IRQ

## DOBRZE

przełącz LED  
ustaw flagę  
zwiększ licznik  
zapisz znacznik czasu

## OSTROŻNIE

długie obliczenia  
sleep()  
transmisja sieciowa  
dużo print()

Callback timera powinien zakończyć się szybko. Dłuższą pracę najlepiej przenieść do głównego toku programu.

# Czy z MicroPythona można odczytać fizyczny TIMER0/TIMER1?

```
from machine import mem32

TIMER0_BASE = 0x400B0000
TIMEHR = 0x08
TIMELR = 0x0C

# najpierw TIMELR - zatrzymuje starsza część
lo = mem32[TIMER0_BASE + TIMELR] & 0xffffffff
hi = mem32[TIMER0_BASE + TIMEHR] & 0xffffffff

time_us = (hi << 32) | lo
print(time_us)
```

- ▶ **Tak, na niskim poziomie:** `machine.mem32` pozwala czytać i zapisywać rejestry peryferiów.
- ▶ TIMER0 ma bazę `0x400B0000`, a TIMER1 `0x400B8000`.
- ▶ Można więc odczytać licznik i konfigurować rejestry alarmów bezpośrednio.
- ▶ To jednak **nie tworzy automatycznie Pythonowego callbacka IRQ**. Pełna obsługa alarmu sprzętowego wymaga znacznie bardziej niskopoziomowego kodu.

Na laboratoriach używamy `machine.Timer`; bezpośrednią kontrolę TIMER0/TIMER1 pokazujemy jako poziom zaawansowany / Pico SDK.

## AON Timer

działa w domenie „always-on”  
może odmierzać dłuższy czas  
może wybudzać układ z trybów niskiego poboru mocy

- ▶ TIMER0/TIMER1 są naturalne dla krótkich, precyzyjnych alarmów i typowego działania programu.
- ▶ AON jest szczególnie użyteczny, gdy układ ma **oszczędzać energię** i mimo to zachować źródło czasu.
- ▶ To nie jest „trzeci taki sam TIMER” — jego rola jest inna.

Które zdanie najlepiej opisuje różnicę między `sleep()` a timerem?

A. Nie ma żadnej różnicy

B. `sleep()` wstrzymuje główny tok kodu, timer może uruchomić callback po czasie

C. Timer działa tylko z GPIO

D. `sleep()` jest sprzętowym przerwaniem

Które zdanie najlepiej opisuje różnicę między `sleep()` a timerem?

A. Nie ma żadnej różnicy

B. `sleep()` wstrzymuje główny tok kodu, timer może uruchomić callback po czasie

C. Timer działa tylko z GPIO

D. `sleep()` jest sprzętowym przerwaniem

Jak najtrafniej opisać `machine.Timer` na porcie MicroPython RP2?

A. Zawsze wybiera bezpośrednio sprzętowy `TIMER0` lub `TIMER1` po numerze

B. Jest timerem wirtualnym zarządzanym przez port MicroPython

C. Jest aliasem funkcji `sleep()`

D. Działa tylko na RPi Pico pierwszej generacji

Jak najtrafniej opisać `machine.Timer` na porcie MicroPython RP2?

A. Zawsze wybiera bezpośrednio sprzętowy `TIMER0` lub `TIMER1` po numerze

**B. Jest timerem wirtualnym zarządzanym przez port MicroPython**

C. Jest aliasem funkcji `sleep()`

D. Działa tylko na RPi Pico pierwszej generacji

# Co powinniśmy umieć po tym wykładzie?

- ▶ skonfigurować GPIO jako wejście i wyjście,
- ▶ wyjaśnić rolę pull-up i pull-down,
- ▶ rozpoznać polling i jego ograniczenia,
- ▶ wyjaśnić drgania styków i debounce,
- ▶ skonfigurować przerwanie na zbocze,
- ▶ napisać krótki handler,
- ▶ odróżnić kod blokujący od nieblokującego,
- ▶ użyć `Timer.PERIODIC` i `Timer.ONE_SHOT`.

## Następny wykład

sygnały analogowe  
ADC ▪ PWM ▪ UART

**LAB** przycisk, IRQ, czas

# Back-up

Dodatkowe materiały dla zainteresowanych

# Alarm wykorzystujący sprzętowy timer w Pico SDK (C/C++)

```
#include "pico/stdlib.h"

int64_t alarm_cb(alarm_id_t id,
void *user_data) {
    // szybka obsługa zdarzenia
    return 0;    // alarm jednorazowy
}

int main() {
    stdio_init_all();

    add_alarm_in_ms(2000, alarm_cb, NULL, false);

    while (true) {
        tight_loop_contents();
    }
}
```

- ▶ `add_alarm_in_ms()` korzysta z infrastruktury alarmów Pico SDK.
- ▶ Callback zostaje wywołany po zadanym czasie.
- ▶ SDK przydziela alarm z puli opartej na sprzętowym timerze.

Na naszym kursie programujemy w MicroPythonie, więc jest to slajd pokazujący, co znajduje się „piętro niżej”. Pico SDK ma również niższy poziom API: `timer_get_instance()` i `timer_hardware_alarm_*`(), gdy chcemy jawnie pracować z konkretnym `TIMER0/TIMER1`.

# Timer okresowy oparty na sprzętowym alarmie w Pico SDK

```
#include "pico/stdlib.h"

bool tick_cb(repeating_timer_t *t) {
    / kod wykonywany okresowo
    return true;    // kontynuuj
}

int main() {
    repeating_timer_t timer;

    add_repeating_timer_ms(500, tick_cb, NULL, &timer);

    while (true) {
        tight_loop_contents();
    }
}
```

- ▶ callback jest wywoływany co 500 ms,
- ▶ zwrócenie true podtrzymuje cykliczne wywołania,
- ▶ false kończy powtarzanie.

Pico SDK daje dużo większą kontrolę nad sprzętem niż standardowe API `machine.Timer`.