# ADC, PWM i UART

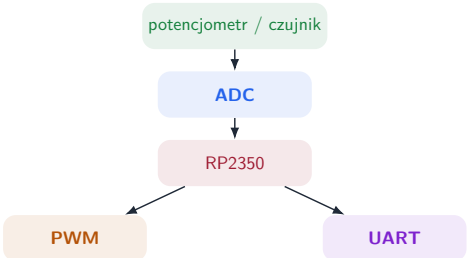
## Od napięcia analogowego do komunikacji szeregowej




Podstawy Systemów Mikroprocesorowych ▪ Wykład 3

dr inż. Dariusz Tefelski

# Na jaki problem dzisiaj odpowiemy?




- Jak zmierzyć napięcie inne niż LOW/HIGH?
- Jak sterować jasnością LED?
- Jak przesłać dane do drugiego układu?

**Plan wykładu:**

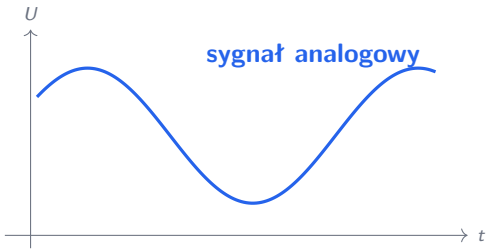
- **ADC** — próbkowanie, kwantyzacja, 12 bitów, `read_u16()`
- **PWM** — okres, częstotliwość, duty cycle, `duty_u16()`
- **UART** — TX/RX, ramka 8N1, baud rate, wysyłanie i odbiór

Od **pomiaru** wielkości analogowej, przez **sterowanie**, do **komunikacji**.

# ADC — od napięcia do liczby

Jak mikrokontroler mierzy świat analogowy?

# Sygnał analogowy a GPIO cyfrowe




Temperatura, napięcie z potencjometru, sygnał z mikrofonu czy czujnika światła mogą przyjmować **wiele wartości pośrednich**.

**GPIO - sygnały cyfrowe**

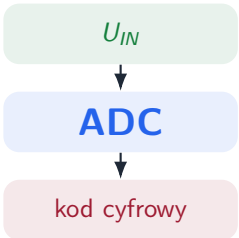
**0 albo 1**

**sygnały analogowe**

**U=(0;3.3) V**

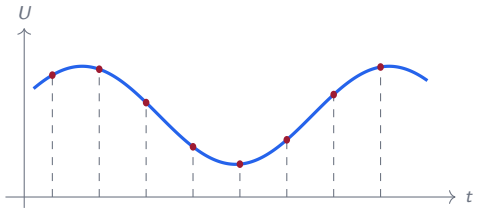
Potrzebujemy układu, który zamieni **napięcie** na **liczbę**.

# ADC: od napięcia analogowego do kolejnych próbek




ADC = *Analog-to-Digital Converter*.

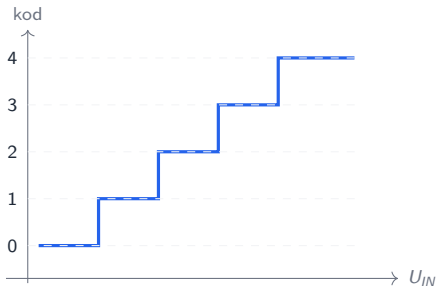
Najpierw wybiera chwilę pomiaru, potem przypisuje wynik do jednego z dostępnych kodów.




**Próbkowanie** = pomiar w kolejnych chwilach czasu.

**Kwantyzacja** = przypisanie zmierzonej wartości do jednego z kodów cyfrowych.

# Kwantyzacja i rozdzielczość ADC




Napięcie jest ciągłe, ale wynik jest liczbą całkowitą — każda próbka trafia do jednego z dostępnych poziomów.

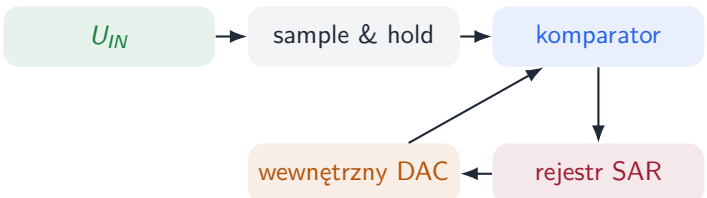
## $2^N$ poziomów

| Bity | Liczba wartości |
|---|---|
| 8 | 256 |
| 10 | 1024 |
| 12 | 4096 |
| 16 | 65536 |

**RP2350: SAR ADC, 12 bitów, do 500 ksps**

sprzętowo 4096 kodów: 0–4095

# Co dzieje się wewnątrz ADC typu SAR?




SAR = *Successive Approximation Register*. Układ nie „mierzy wszystkich 4096 poziomów po kolei" — szuka kodu **bit po bicie**.

Inne spotykane architektury ADC to m.in. FLASH, całkujące i $\Delta\Sigma$; RP2350 wykorzystuje SAR.

1. próbka napięcia zostaje zatrzymana,
2. SAR ustawia próbny kod,
3. wewnętrzny DAC zamienia go na napięcie,
4. komparator sprawdza: za wysoko czy za nisko?
5. po decyzji dla kolejnych bitów otrzymujemy kod końcowy.

# SAR: wyszukiwanie wyniku od najstarszego bitu

Przykład uproszczony dla **4-bitowego ADC** i $U_{IN} \approx 0{,}68\ V_{REF}$:

| Krok | Próbowany kod | $U_{DAC}$ | Decyzja |
|---|---|---|---|
| 1 | 1000 | $0{,}50\ V_{REF}$ | za nisko $\Rightarrow$ bit zostaje |
| 2 | 1100 | $0{,}75\ V_{REF}$ | za wysoko $\Rightarrow$ bit kasujemy |
| 3 | 1010 | $0{,}625\ V_{REF}$ | za nisko $\Rightarrow$ bit zostaje |
| 4 | 1011 | $0{,}6875\ V_{REF}$ | lekko za wysoko $\Rightarrow$ bit kasujemy |

Wynik: $1010_2 = 10$.

Dla 12 bitów zasada jest taka sama — komparator wykonuje kolejne decyzje od MSB do LSB.

# Ile napięcia odpowiada jednemu krokowi?

Dla idealnego ADC $N$-bitowego:

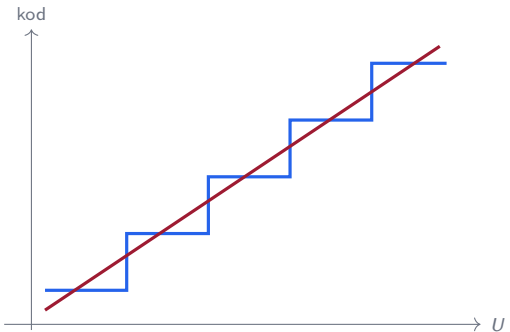
$$\Delta U \approx \frac{V_{REF}}{2^N}$$

Dla $N = 12$ i $V_{REF} \approx 3{,}3$ V:

$$\Delta U \approx \frac{3{,}3}{4096} \approx 0{,}806\ \mathrm{mV}$$

To jest idealny krok kwantyzacji. Rzeczywisty pomiar ma dodatkowo szum i błędy układu analogowego.

# Błąd kwantyzacji: nie da się zapisać każdej wartości idealnie




W idealnym modelu błąd zaokrąglenia jest rzędu:

$$\pm\frac{1}{2}\,LSB$$

- wynik jest dyskretny,
- napięcie wejściowe ciągłe,
- najbliższy kod jest przybliżeniem.

# Które piny RPi Pico 2 W są analogowe?




GP26–GP28 mogą pracować jako wejścia ADC. Ten sam pin można też skonfigurować jako GPIO cyfrowe.

**Wewnętrzny czujnik temperatury** jest podłączony do osobnego wejścia ADC. W MicroPythonie korzystamy z niego tak:

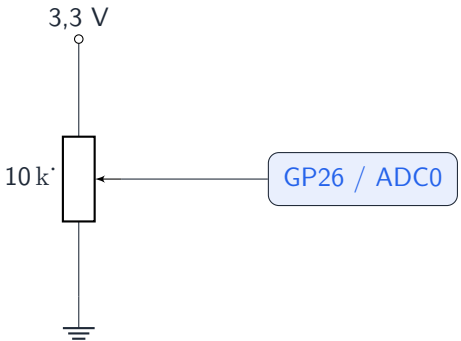
```
from machine import ADC

temp_adc = ADC(ADC.CORE_TEMP)
raw = temp_adc.read_u16()
voltage = raw * 3.3 / 65535

temp_c = 27 - (voltage - 0.706) / 0.001721
print(temp_c)
```

To jest temperatura **struktury układu**, nie temperatura powietrza. Wynik jest orientacyjny i zależy m.in. od dokładności napięcia odniesienia.

# Potencjometr jako źródło napięcia dla ADC




Skrajne wyprowadzenia potencjometru są między 3,3 V i GND, a **suwak** daje napięcie od około 0 do 3,3 V.

```
from machine import ADC, Pin

adc = ADC(Pin(26))

value = adc.read_u16()
print(value)
```

`read_u16()` zwraca wynik w zunifikowanej skali **0–65535**; nie oznacza to, że sprzętowy ADC ma 16 bitów.

# Ważne: `read_u16()` nie oznacza 16-bitowego ADC

**sprzęt**

12 bitów

4096 poziomów

**API MicroPython**

0–65535

wynik przeskalowany do 16 bitów

Większy zakres liczbowy API **nie zwiększa fizycznej rozdzielczości ADC**.

# Jak przeliczyć `read_u16()` na napięcie?

Dla przybliżonego zakresu 0–3,3 V:

$$U \approx \frac{\texttt{value}}{65535} \cdot 3{,}3\ \mathrm{V}$$

Przykład dla `value = 32768`:

$$U \approx \frac{32768}{65535} \cdot 3{,}3 \approx 1{,}65\ \mathrm{V}$$

W pomiarach dokładnościowych należałoby znać rzeczywiste napięcie odniesienia, a nie zakładać idealne 3,300 V.

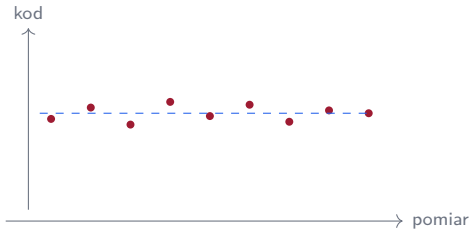
# Pełny przykład: potencjometr i napięcie

```
from machine import ADC, Pin
from time import sleep_ms

adc = ADC(Pin(26))

while True:
    raw = adc.read_u16()
    voltage = raw * 3.3 / 65535
    print(raw, voltage)
    sleep_ms(200)
```

# Wahania odczytu i uśrednianie próbek




Wahania mogą wynikać z szumu, zasilania/odniesienia, zakłóceń części cyfrowej i kwantyzacji.

```
sum_value = 0

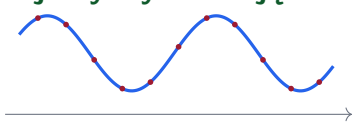
for i in range(16):
    sum_value += adc.read_u16()

average = sum_value / 16
print(average)
```

Uśrednianie ogranicza losowe wahania, ale **nie usuwa błędów systematycznych** i spowalnia reakcję pomiaru.

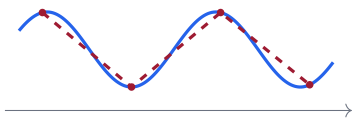
# Częstotliwość próbkowania, aliasing i warunek Nyquista

**próbkujemy wystarczająco często**



wiele próbek dobrze opisuje kształt przebiegu

**próbkujemy zbyt rzadko**



próbki mogą sugerować inny, fałszywy przebieg — **aliasing**

Dla sygnału ograniczonego do $f_{max}$ potrzebujemy co najmniej

$$\boxed{f_s > 2f_{max}}$$

w praktyce zwykle próbkujemy wyraźnie szybciej niż to minimum.

# Pomiar napięcia: o czym trzeba pamiętać?

- **Nie przekraczamy dopuszczalnego zakresu wejścia ADC.**
- Źródło sygnału musi mieć wspólną masę z RPi Pico 2 W.
- Napięcie odniesienia i zasilanie wpływają na dokładność.
- Pojedyncza próbka może się wahać — czasem lepiej użyć uśredniania.
- Częstotliwość próbkowania dobieramy do dynamiki mierzonego sygnału.

# Czas na podsumowanie tej części wykładu!

ADC ▪ próbkowanie ▪ kwantyzacja

# Pytanie kontrolne 1

Ile różnych wartości może zwrócić 12-bitowy ADC?

A. 12

B. 1024

C. 4096

D. 65535

# Pytanie kontrolne 1

Ile różnych wartości może zwrócić 12-bitowy ADC?

A. 12

B. 1024

**C. 4096**

D. 65535

# Pytanie kontrolne 2

Co oznacza fakt, że `read_u16()` zwraca wartości z zakresu 0–65535?

A. RP2350 ma fizyczny ADC 16-bitowy

B. MicroPython przeskalowuje wynik sprzętowego ADC do zakresu 16-bitowego

C. ADC mierzy tylko 0 lub 65535

D. Funkcja działa tylko z PWM

# Pytanie kontrolne 2

Co oznacza fakt, że `read_u16()` zwraca wartości z zakresu 0–65535?

A. RP2350 ma fizyczny ADC 16-bitowy

**B. MicroPython przeskalowuje wynik sprzętowego ADC do zakresu 16-bitowego**

C. ADC mierzy tylko 0 lub 65535

D. Funkcja działa tylko z PWM

# Pytanie kontrolne 3

Co może się stać, gdy sygnał zmienia się szybko, a próbkujemy go zbyt rzadko?

A. Pojawi się aliasing i możemy błędnie ocenić przebieg

B. ADC automatycznie zwiększy rozdzielczość

C. Napięcie odniesienia wzrośnie

D. UART przejmie pomiar

# Pytanie kontrolne 3

Co może się stać, gdy sygnał zmienia się szybko, a próbkujemy go zbyt rzadko?

**A. Pojawi się aliasing i możemy błędnie ocenić przebieg**

B. ADC automatycznie zwiększy rozdzielczość

C. Napięcie odniesienia wzrośnie

D. UART przejmie pomiar

# PWM — sterowanie wypełnieniem

Jak uzyskać „średni efekt” z wyjścia cyfrowego?

# Problem: GPIO zna tylko LOW i HIGH

LOW $\approx$ 0 V

?

HIGH $\approx$ 3,3 V

Jak więc **regulować jasność LED**, skoro pin ma tylko dwa poziomy napięcia?

# PWM: okres, częstotliwość i wypełnienie

PWM przełącza sygnał między LOW i HIGH. Informację przenosi **czas trwania stanu HIGH**.

**25%**



stan HIGH przez $0{,}25\,T$

**50%**




stan HIGH przez $0{,}50\,T$

**75%**



stan HIGH przez $0{,}75\,T$

$$D = \frac{t_{HIGH}}{T} \cdot 100\% \qquad f = \frac{1}{T}$$

Większy duty cycle = większa część każdego okresu spędzona w stanie HIGH.

# Średnia wartość

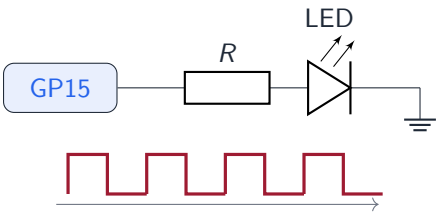
Dla idealnego PWM 0–$V_{CC}$:

$$\overline{U} \approx D \cdot V_{CC}$$

| Duty cycle | wartość średnia dla 3,3 V |
|---|---|
| 25% | $\approx$ 0,83 V |
| 50% | $\approx$ 1,65 V |
| 75% | $\approx$ 2,48 V |

To wartość średnia, nie informacja o chwilowym napięciu na pinie. Natomiast można dodać filtr RC by uzyskać sygnał analogowy (więcej na kolejnych slajdach).

# PWM w praktyce: sterowanie jasnością LED




LED jest szybko włączana i wyłączana; oko odbiera średnią jasność.

```
from machine import Pin, PWM

led = PWM(Pin(15))
led.freq(1000)
led.duty_u16(32768)
```

- `freq(1000)` — 1000 okresów/s,
- `duty_u16(32768)` — 32768/65535 $\approx$ 50% ,
- pin nadal przełącza się między LOW i HIGH.

# `duty_u16()` — wygodna skala 0–65535




API używa zakresu 16-bitowego jako wygodnej skali wypełnienia.

# Przykład: płynne rozjaśnianie i ściemnianie LED

```
from machine import Pin, PWM
from time import sleep_ms

led = PWM(Pin(15), freq=1000)

while True:
    for duty in range(0, 65536, 1024):
        led.duty_u16(duty)
        sleep_ms(20)

    for duty in range(65535, -1, -1024):
        led.duty_u16(duty)
        sleep_ms(20)
```

`range(start, stop, step)` ma trzy argumenty:

- `start` — wartość początkowa,
- `stop` — granica (bez tej wartości),
- `step` — krok zmiany.

Pierwsza pętla zwiększa duty, druga zmniejsza je krokiem 1024.

# Co zmieniamy: częstotliwość czy duty cycle?

## częstotliwość

- jak szybko powtarza się przebieg,
- $f = 1/T$,
- wpływa na słyszalność, migotanie i pracę odbiornika.

## duty cycle

- jaka część okresu jest HIGH,
- decyduje o średnim efekcie,
- zmienia np. jasność LED.

# Dobór częstotliwości PWM a rozdzielczość

**Jak dobierać $f$?**

- LED: na tyle wysoko, aby nie było widocznego migotania,
- silnik: zgodnie z wymaganiami sterownika i napędu,
- serwo: istotne są zadany okres i szerokość impulsu.

**większa częstotliwość**

$\Downarrow$

**mniej kroków licznika**

Sprzęt ma skończony zegar i licznik, więc częstotliwość i efektywna rozdzielczość duty cycle są współzależne.

# PWM w RP2350 i mapowanie pinów RPi Pico 2 W

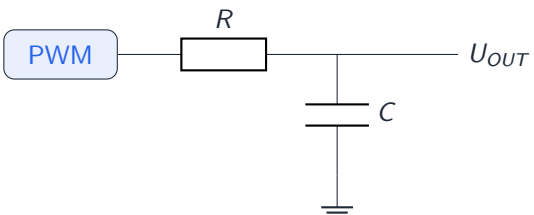
- RP2350 ma **12 bloków PWM**, każdy z kanałami A i B.
- Na pinach wyprowadzonych w RPi Pico 2 W dostępne są mapowania do slice 0–7.
- Kanały A i B **tego samego slice współdzielą częstotliwość**, ale mają niezależne wypełnienie.
- Kilka GPIO może wskazywać na ten sam slice/kanał.

Przykład: GP0 i GP16 → PWM0 A, GP1 i GP17 → PWM0 B.

| slice | kanał A | kanał B |
|---|---|---|
| 0 | GP0 / GP16 | GP1 / GP17 |
| 1 | GP2 / GP18 | GP3 / GP19 |
| 2 | GP4 / GP20 | GP5 / GP21 |
| 3 | GP6 / GP22 | GP7 |
| 4 | GP8 | GP9 |
| 5 | GP10 / GP26 | GP11 / GP27 |
| 6 | GP12 / GP28 | GP13 |
| 7 | GP14 | GP15 |

**pinout PWM:** pary A/B należą do jednego generatora (slice)

# Filtr RC może wygładzić PWM

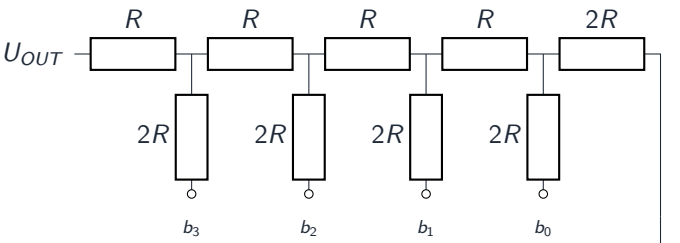



Filtr dolnoprzepustowy tłumi szybkie przełączenia i pozostawia wolnozmienną składową średnią.

- większe duty cycle $\Rightarrow$ większe średnie $U_{OUT}$,
- na wyjściu pozostają tętnienia,
- szybkość reakcji zależy od stałej czasowej $RC$,
- to nadal **nie jest idealny DAC**.

# Jak DAC zamienia kod cyfrowy na napięcie?

**Przykład: 4-bitowa drabinka R–2R**




każdy bit steruje przełączeniem swojego węzła między 0 V i $V_{REF}$

Drabinka używa tylko dwóch wartości rezystorów: $R$ i $2R$. Kolejne bity mają wagi malejące dwukrotnie.

Dla prostej 4-bitowej drabinki:

$$U_{OUT} = V_{REF}\left(\frac{b_3}{2} + \frac{b_2}{4} + \frac{b_1}{8} + \frac{b_0}{16}\right)$$

Dla 1010:

$$U_{OUT} = V_{REF}\left(\frac{1}{2} + \frac{1}{8}\right) = 0{,}625\,V_{REF}$$

W praktycznym DAC ważne są także: **rozdzielczość**, **czas ustalania** i **monotoniczność**. Układ scalony może dodatkowo buforować lub skalować wyjście.

# PWM a prawdziwy DAC

## PWM

- pin przełącza LOW/HIGH,
- informacja jest w wypełnieniu,
- filtr RC może utworzyć przybliżone napięcie analogowe,
- dobre do LED i wielu układów sterujących.

## DAC

- kod bezpośrednio wybiera poziom napięcia/prądu,
- kolejne kody tworzą schodkowy sygnał analogowy,
- zwykle potrzebny jest bufor/filtr wyjściowy,
- RP2350 nie ma klasycznego ogólnego DAC napięciowego.

Gdy potrzebujemy prawdziwego wyjścia analogowego, możemy użyć **zewnętrznego DAC** sterowanego np. przez $I^2C$ lub SPI. Więcej na ten temat na laboratoriach.

# Czas na podsumowanie tej części wykładu!

PWM ▪ częstotliwość ▪ duty cycle

# Pytanie kontrolne 4

Które zdanie o PWM przy wypełnieniu 50% jest poprawne?

A. Na pinie cały czas jest dokładnie 1,65 V

B. Pin przełącza się między LOW i HIGH, a HIGH trwa około połowy okresu

C. PWM zamienia GPIO w wejście analogowe

D. Częstotliwość zawsze wynosi 50 Hz

# Pytanie kontrolne 4

Które zdanie o PWM przy wypełnieniu 50% jest poprawne?

A. Na pinie cały czas jest dokładnie 1,65 V

**B. Pin przełącza się między LOW i HIGH, a HIGH trwa około połowy okresu**

C. PWM zamienia GPIO w wejście analogowe

D. Częstotliwość zawsze wynosi 50 Hz

# Pytanie kontrolne 5

Jaka wartość `duty_u16()` odpowiada w przybliżeniu 50% wypełnienia?

A. 0

B. 16384

C. 32768

D. 65535

# Pytanie kontrolne 5

Jaka wartość `duty_u16()` odpowiada w przybliżeniu 50% wypełnienia?

A. 0

B. 16384

**C. 32768**

D. 65535

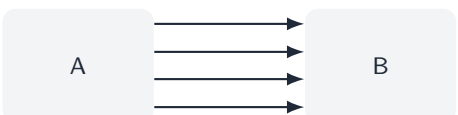
# UART — komunikacja szeregowa

Jak dwa urządzenia przesyłają bajty bez wspólnej linii zegara?

# Magistrale komunikacyjne: równolegle czy szeregowo?

**Magistrala/interfejs komunikacyjny** określa, jak układy fizycznie przesyłają bity i jak interpretują kolejne sygnały.

**równolegle**




wiele linii; kilka bitów może być przesyłanych równocześnie

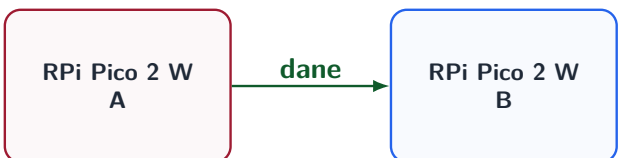
**szeregowo**




bity idą po kolei; mniej przewodów i łatwiejsze prowadzenie połączeń

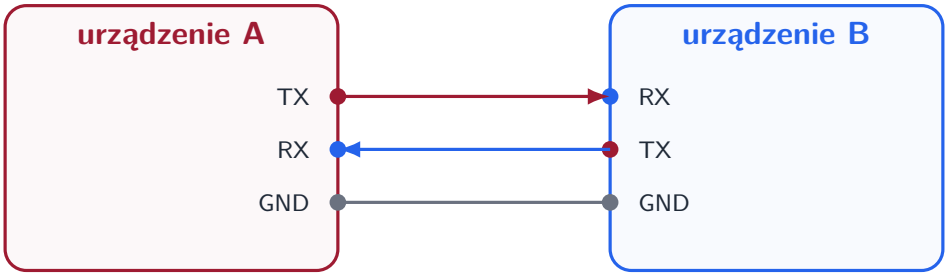
Przykłady komunikacji szeregowej: **UART, I$^2$C, SPI, 1-Wire**. Dzisiaj skupiamy się na UART.

# Po co nam komunikacja szeregowa?




- wyniki pomiarów,
- konfiguracja modułów,
- debugowanie,
- komunikacja między MCU i czujnikami.

# UART: TX, RX i wspólna masa

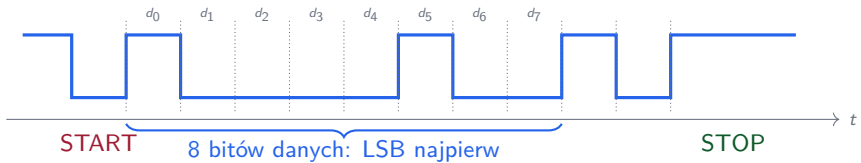



**TX → RX**
nadajnik jednej strony do odbiornika drugiej

**wspólne GND**
wspólny poziom odniesienia napięć

**brak CLK**
UART jest asynchroniczny; czas bitu wynika z baud rate

# Ramka UART: przykład 8N1 i kolejność bitów




**8N1** oznacza:

- 8 bitów danych,
- N = bez bitu parzystości,
- 1 bit stopu,
- razem zwykle 10 czasów bitu na jeden bajt.

Dla znaku `'A'`:

$$0100\,0001_2 = 0x41$$

UART wysyła dane od bitu najmłodszego, więc pierwszym bitem danych jest **1**.

# Baud rate: jak szybko przesyłamy bity?

Baud rate w typowym UART odpowiada liczbie symboli/bitów przesyłanych w ciągu sekundy.

$$T_{bit} = \frac{1}{\text{baud rate}}$$

Dla 115200 baud:

$$T_{bit} \approx 8{,}68\ \mu\text{s}$$

Jedna ramka 8N1 (10 bitów) trwa około 86,8 $\mu$s.

# Parity — prosty mechanizm wykrywania części błędów

**parity**

dodatkowy bit dobrany na podstawie liczby jedynek

- brak parity: `None`,
- parzystość parzysta lub nieparzysta,
- pomaga wykryć niektóre błędy,
- nie jest pełnym mechanizmem kontroli integralności danych.

# MicroPython: konfiguracja i wysyłanie przez UART

```
from machine import UART, Pin

uart = UART(
    0,
    baudrate=115200,
    tx=Pin(0),
    rx=Pin(1),
    bits=8,
    parity=None,
    stop=1
)

uart.write(b'Hello\n')
```

- UART0,
- 115200 bit/s,
- 8N1,
- GP0 jako TX,
- GP1 jako RX.

`write()` wysyła bufor bajtów i zwraca liczbę wysłanych bajtów (lub `None` przy timeout).

# Odbieranie danych: `any()`, `read()` i `readline()`

```
if uart.any():
    data = uart.read()
    print(data)
```

`any()` informuje, czy dane są dostępne bez blokowania.

| Wywołanie | Znaczenie |
|---|---|
| `read()` | odczytaj dostępne bajty |
| `read(5)` | odczytaj najwyżej 5 bajtów |
| `readline()` | odczytaj do końca linii |

W komunikacji tekstowej wygodnie kończyć komunikaty znakiem `\n`.

# Pełny nadajnik: wysyłamy licznik

```
from machine import UART, Pin
from time import sleep_ms

uart = UART(0, 115200, tx=Pin(0), rx=Pin(1))

counter = 0
while True:
    uart.write("count={}\n".format(counter))
    counter += 1
    sleep_ms(500)
```

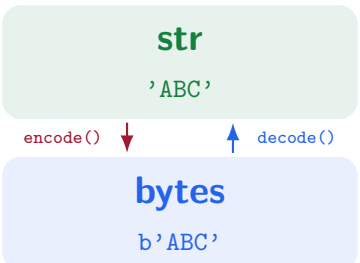
# Pełny odbiornik: czytamy całe linie

```
from machine import UART, Pin

uart = UART(0, 115200, tx=Pin(0), rx=Pin(1))

while True:
    if uart.any():
        line = uart.readline()
        if line is not None:
            print(line)
```

# Bajty i tekst: `encode()` i `decode()`




```
text = 'ABC'

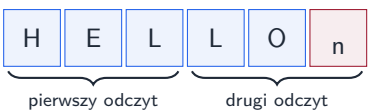
data = text.encode('utf-8')
# data == b'ABC'

decoded_text = data.decode('utf-8')
print(decoded_text)

uart.write(data)
```

UART przesyła **bajty**; `encode()` zamienia tekst na bajty, a `decode()` bajty z powrotem na tekst.

# Odbiór może przyjść fragmentami




- UART jest strumieniem bajtów,
- jeden `read()` nie musi odpowiadać jednemu „komunikatowi”,
- protokół powinien określać granice danych.

# Najczęstsze problemy z UART

- TX połączony z TX zamiast TX z RX.
- Brak wspólnej masy.
- Inny baud rate po obu stronach.
- Niezgodne parametry ramki (bits/parity/stop).
- Brak znaku końca linii mimo użycia `readline()`.
- Próba interpretacji surowych bajtów jako tekstu bez sprawdzenia kodowania.
- Zbyt długie blokowanie programu pomiędzy kolejnymi odczytami.

# UART a USB/REPL — nie mylmy tych kanałów

## USB / REPL

- połączenie z komputerem przez USB,
- konsola MicroPythona,
- `print()` widoczny w terminalu IDE.

## UART

- osobne sprzętowe peryferium,
- piny TX/RX,
- komunikacja z drugim urządzeniem.

# Czas na podsumowanie tej części wykładu!

UART ▪ TX/RX ▪ 8N1 ▪ baud rate

# Pytanie kontrolne 6

Jak prawidłowo połączyć dwa urządzenia UART do komunikacji dwukierunkowej?

A. TX–TX i RX–RX

B. TX A–RX B, RX A–TX B oraz wspólne GND

C. Tylko jeden przewód TX

D. Wyłącznie linie GND bez TX/RX

# Pytanie kontrolne 6

Jak prawidłowo połączyć dwa urządzenia UART do komunikacji dwukierunkowej?

A. TX–TX i RX–RX

**B. TX A–RX B, RX A–TX B oraz wspólne GND**

C. Tylko jeden przewód TX

D. Wyłącznie linie GND bez TX/RX

# Pytanie kontrolne 7

Co oznacza zapis 8N1 w typowej konfiguracji UART?

A. 8 UART-ów, brak napięcia, 1 przewód

B. 8 bitów danych, bez parity, 1 bit stopu

C. 8 bitów startu, normalny zegar, 1 bit danych

D. 8 MHz, brak RX, 1 bit stopu

# Pytanie kontrolne 7

Co oznacza zapis 8N1 w typowej konfiguracji UART?

A. 8 UART-ów, brak napięcia, 1 przewód

**B. 8 bitów danych, bez parity, 1 bit stopu**

C. 8 bitów startu, normalny zegar, 1 bit danych

D. 8 MHz, brak RX, 1 bit stopu

# Pytanie kontrolne 8

Co najczęściej stanie się, gdy jedna strona pracuje z 9600 baud, a druga z 115200 baud?

A. Odbiór będzie błędny lub nieczytelny

B. UART automatycznie uzgodni szybkość

C. ADC zmieni rozdzielczość

D. PWM ustawi 50% duty

# Pytanie kontrolne 8

Co najczęściej stanie się, gdy jedna strona pracuje z 9600 baud, a druga z 115200 baud?

**A. Odbiór będzie błędny lub nieczytelny**

B. UART automatycznie uzgodni szybkość

C. ADC zmieni rozdzielczość

D. PWM ustawi 50% duty

# Co powinniśmy umieć po Wykładzie 3?

- wyjaśnić próbkowanie i kwantyzację,
- przeliczyć rozdzielczość ADC i krok LSB,
- użyć `ADC.read_u16()`,
- wyjaśnić aliasing na poziomie intuicyjnym,
- skonfigurować PWM i duty cycle,
- odróżnić PWM od prawdziwego DAC,
- połączyć TX/RX/GND,
- skonfigurować UART i odebrać dane.

**Następny wykład**

komunikacja cyfrowa
$I^2C$ ▪ SPI ▪ 1-Wire