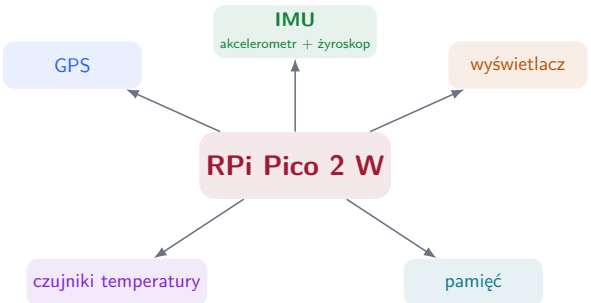
Magistrale komunikacyjne

UART, I^{2}C, SPI i 1-Wire



Podstawy Systemów Mikroprocesorowych ▪ Wykład 4
dr inż. Dariusz Tefelski

Jeden mikrokontroler, wiele urządzeń — jak je połączyć?



IMU (*Inertial Measurement Unit*) mierzy ruch/orientację; typowy moduł łączy akcelerometr i żyroskop.

Te same dane można przesyłać różnymi interfejsami, ale każdy z nich realizowany jest w inny sposób.

Dzisiaj pytamy przede wszystkim:

- ile przewodów jest potrzebnych?
- kto nadaje zegar?
- jak wybieramy urządzenie?
- czy wiele urządzeń może współdzielić linie?

Nie uczymy się czterech list komend. Uczymy się **czterech sposobów organizacji komunikacji**.

Co właściwie opisuje interfejs komunikacyjny?

Warstwa elektryczna

- liczba i rola przewodów,
- poziomy logiczne i sposób sterowania linią,
- wspólna masa,
- czy potrzebne są rezystory podciągające,
- maksymalna szybkość zależna od sprzętu i połączeń.

Organizacja transmisji

- synchroniczna czy asynchroniczna,
- kto inicjuje transmisję,
- czy możliwa jest komunikacja w obu kierunkach.

Protokół

- gdzie zaczyna się i kończy transmisja,
- jak wygląda bajt/ramka,
- czy istnieje adres urządzenia,
- jak odbiornik potwierdza dane,
- co oznaczają kolejne bajty.

Ważne: ten sam przewód może poprawnie przenosić zera i jedynki, a komunikacja nadal nie zadziała, jeśli urządzenia nie zgadzają się co do **protokołu**.

Transmisja równoległa i szeregowa



wiele bitów jednocześnie, ale dużo przewodów



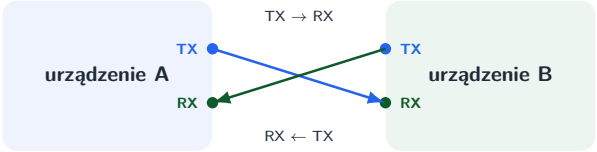
bity kolejno w czasie, mniej przewodów

UART, I^2C, SPI i 1-Wire to różne sposoby organizacji **transmisji szeregowej**.

Od punkt–punkt do magistrali

UART jako punkt odniesienia

UART — co już wiemy i czego UART nie rozwiązuje?



Dwa niezależne kierunki, bez wspólnego zegara. Obie strony muszą znać baud rate i format ramki.

UART jest świetny, gdy:

- łączymy dwa urządzenia,
- każde ma TX i RX,
- dane są strumieniem bajtów.

Ale sam UART nie definiuje:

- adresowania wielu urządzeń,
- współdzielenia jednej pary linii przez wiele peryferiów,
- wspólnego zegara transmisji.

Sprzętowy kontroler czy bit-banging?

sprzętowy interfejs

- CPU ustawia parametry i przekazuje dane,
- generowanie zegara/bitów realizuje sprzęt,
- mniejsze obciążenie procesora,
- lepsza powtarzalność czasowa.

RP2350 ma sprzętowo po **2 kontrolery UART, SPI i I^2C**.

bit-banging

Bit-banging to programowa emulacja interfejsu: kod sam ustawia i odczytuje GPIO w odpowiednich chwilach, zamiast korzystać z dedykowanego kontrolera sprzętowego.

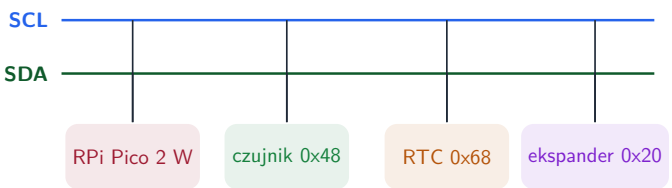
- większa swoboda wyboru pinów,
- większe obciążenie CPU i większa zależność od czasu wykonania kodu,
- przydatne, gdy brak odpowiedniego bloku sprzętowego.

MicroPython udostępnia m.in. `SoftI2C` i `SoftSPI`; sterownik 1-Wire jest programowy.

I^2C

Dwie linie, wiele urządzeń, adresowanie

I²C: jedna para linii dla wielu urządzeń

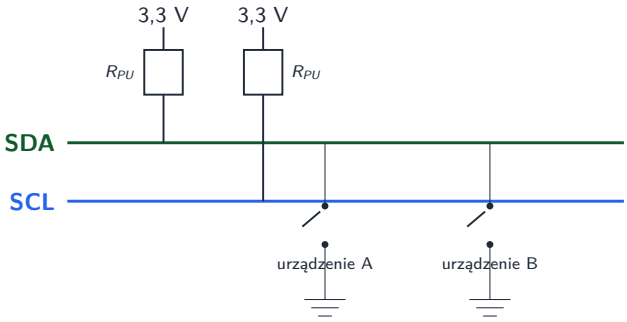


Linie są wspólne: **SCL** niesie zegar, a **SDA** dane i adresy.

- urządzenia są rozróżniane przez **adres**,
- kontroler rozpoczyna transmisję i generuje SCL,
- SDA jest dwukierunkowa,
- typowa konfiguracja ma jednego kontrolera i wiele peryferiów,
- wszystkie urządzenia muszą mieć wspólną masę.

W starszej dokumentacji spotkasz nazwy *master/slave*; obecnie częściej używa się *controller/target* lub *controller/peripheral*.

Dlaczego I²C potrzebuje rezystorów pull-up?



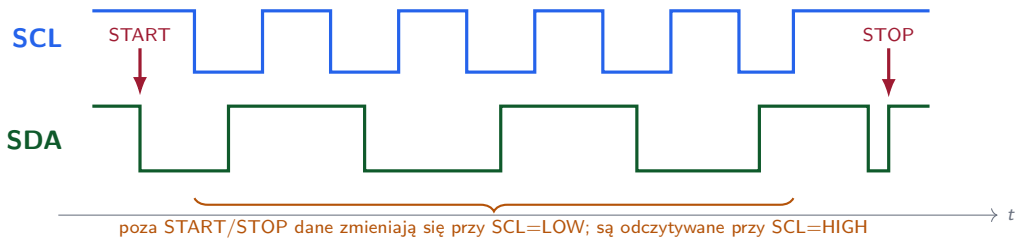
SCL ma taką samą elektrykę open-drain + pull-up. Kontroler zwykle generuje zegar; niektóre peryferia mogą czasowo przytrzymać SCL w LOW (*clock stretching*).

Wyjścia I²C są typu **open-drain**:

- urządzenie potrafi aktywnie wymusić **LOW**,
- stan **HIGH** powstaje, gdy wszystkie urządzenia puszczą linię,
- pull-up przywraca wtedy poziom 3,3 V,
- kilka urządzeń może współdzielić przewód bez zwarcia dwóch przeciwnie ustawionych wyjść.

LOW ma „pierwszeństwo". Dzięki temu można potwierdzać odbiór (ACK), a w układach z kilkoma kontrolerami także **rozstrzygać konflikt**, który z nich może dalej nadawać (tzw. arbitraż).

START, STOP i zasada zmiany danych



START: SDA przechodzi HIGH→LOW, gdy **SCL pozostaje HIGH**. To jednoznacznie oznacza początek transakcji.

STOP: SDA przechodzi LOW→HIGH, gdy **SCL pozostaje HIGH**. To kończy transakcję i zwalnia magistralę.

Bajt I²C ma dziewiąty takt: ACK/NACK

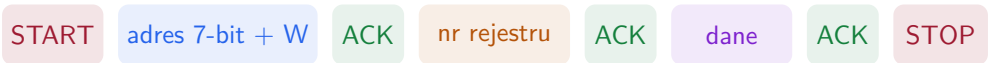


Po każdych 8 bitach nadający **puszcza SDA** na dziewiąty takt zegara.

- **ACK**: odbiornik ściąga SDA do LOW — „odebrałem bajt".
- **NACK**: SDA pozostaje HIGH — brak potwierdzenia albo świadome zakończenie odczytu.

ACK nie mówi, że „wartość była sensowna". Potwierdza poprawne przyjęcie bajtu na poziomie magistrali.

Pełna transakcja I^{2}C: zapis jednego bajtu do rejestru



Znaczenie pól:

- adres wybiera urządzenie,
- bit W oznacza zapis,
- kolejny bajt często wskazuje rejestr,
- następny bajt jest wartością zapisywaną do tego rejestru.

Przykład: czujnik o adresie 0x48

Chcemy wpisać 0x60 do rejestru 0x01:

1. START i bajt adresowy 0x90 $\rightarrow$ ACK,
2. numer rejestru 0x01 $\rightarrow$ ACK,
3. wartość 0x60 $\rightarrow$ ACK,
4. STOP.

`0x90 = (0x48 << 1) | 0;` ostatni bit to **W=0**.

Odczyt rejestru: po co jest repeated START?

START | adres + W | ACK | rejestr | ACK | reSTART | adres + R | ACK | dane | NACK | STOP

Dlaczego nie kończymy transmisji po podaniu numeru rejestru?

1. Kontroler rozpoczyna od kierunku **W**, bo najpierw musi przekazać adres urządzenia i **numer rejestru**, który chce odczytać.
2. Urządzenie zapamiętuje ten numer jako bieżący adres wewnętrzny (ustawia swój wskaźnik rejestru).
3. Kontroler **nie wysyła STOP**: magistrala pozostaje zajęta i żadne inne zdarzenie nie rozdziela obu części operacji.
4. Wysyła **repeated START**, ponownie adresuje to samo urządzenie, tym razem z bitem **R**, a urządzenie zaczyna nadawać zawartość wskazanego rejestru.

Przykład: urządzenie 0x48, rejestr 0x00

1. START, 0x90 (W=0) → ACK,
2. 0x00 – numer rejestru → ACK,
3. repeated START, 0x91 (R=1) → ACK,
4. odczyt DATA_H → ACK,
5. odczyt DATA_L → NACK, potem STOP.

Repeated START łączy „wybierz rejestr” i „odczytaj dane” w **jedną logiczną transakcję**.

I²C w MicroPythonie: najpierw znajdź urządzenia

```
from machine import Pin, I2C

i2c = I2C(
    0,
    scl=Pin(5),
    sda=Pin(4),
    freq=400_000
)

print(i2c.scan())
```

`scan()` wysyła próby adresowania i zwraca listę urządzeń, które odpowiedziały ACK. Przykładowo:

$$[0x3C,\ 0x48,\ 0x68]$$

Skanowanie mówi: **„kto odpowiada?”**. Nie mówi automatycznie, **co to za układ** — ten sam adres może być używany przez różne rodziny urządzeń.

I²C w MicroPythonie: dane i rejestry

```
address = 0x48
register = 0x01

# odczytaj 2 bajty z rejestru
raw = i2c.readfrom_mem(
    address, register, 2
)

# zapisz jeden bajt do rejestru
i2c.writeto_mem(
    address, register, b'\x20'
)
```

Trzy poziomy informacji:

- `address` — które urządzenie,
- `register` — który rejestr w urządzeniu,
- przesyłane bajty — wartość rejestru lub dane pomiarowe.

Adresy rejestrów, kolejność bajtów i znaczenie bitów zawsze sprawdzamy w **datasheet konkretnego układu**.

Przykład: od `scan()` do odczytu rejestru

```
devices = i2c.scan()
print(devices) # [0x48]

raw = i2c.readfrom_mem(
    0x48, # adres urzadzenia
    0x00, # numer rejestru
    2 # liczba bajtow
)

print(raw.hex())
```

Co dzieje się na magistrali?

1. `scan()` sprawdza, czy 0x48 odpowiada ACK,
2. zapisujemy do urządzenia numer rejestru 0x00,
3. repeated START zmienia kierunek,
4. urządzenie odsyła dwa bajty,
5. dopiero datasheet mówi, jak te dwa bajty zinterpretować.

`readfrom_mem()` ukrywa sekwencję START → adres+W → rejestr → reSTART → adres+R → dane.

I²C w praktyce: co najczęściej nie działa?

Elektryka

- brak wspólnej masy,
- brak pull-upów albo niewłaściwa ich wartość,
- zbyt duża pojemność/długość przewodów,
- podciągnięcie magistrali do niewłaściwego napięcia,
- dwa urządzenia o tym samym, niezmienialnym adresie.

Program/protokół

- wybrany zły kontroler I²C lub złe GPIO,
- zły adres 7-bitowy,
- pomylenie adresu urządzenia z adresem rejestru,
- zła liczba bajtów lub kolejność endian,
- urządzenie wymaga konfiguracji przed pierwszym pomiarem.

Pierwszy krok diagnostyczny: `i2c.scan()`. Jeśli urządzenia nie ma na liście, najpierw sprawdzamy **połączenia i zasilanie**, a nie wzór przeliczający wynik.

Pytanie kontrolne 1

Dlaczego linie SDA i SCL w I^2C wymagają rezystorów pull-up?

A. Ponieważ I^2C działa zawsze z napięciem 5 V

B. Ponieważ urządzenia aktywnie wymuszają LOW, a stan HIGH powstaje po zwolnieniu linii

C. Aby zwiększyć liczbę bitów adresu

D. Tylko po to, aby zmniejszyć pobór prądu mikrokontrolera

Pytanie kontrolne 1

Dlaczego linie SDA i SCL w I^2C wymagają rezystorów pull-up?

A. Ponieważ I^2C działa zawsze z napięciem 5 V

B. Ponieważ urządzenia aktywnie wymuszają LOW, a stan HIGH powstaje po zwolnieniu linii

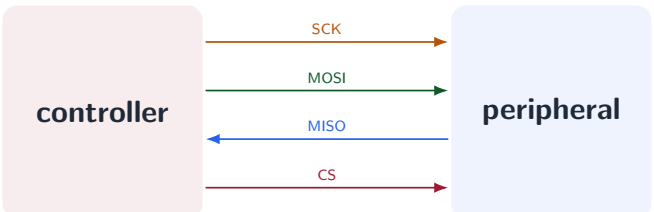
C. Aby zwiększyć liczbę bitów adresu

D. Tylko po to, aby zmniejszyć pobór prądu mikrokontrolera

SPI

Wspólny zegar, szybka transmisja, osobny CS

SPI: cztery sygnały i brak adresu w protokole

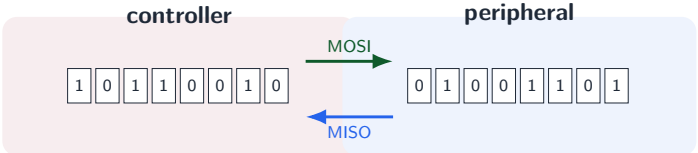


MOSI: controller → peripheral; MISO: peripheral → controller.

- **SCK** — zegar generowany przez controller,
- **MOSI** — dane w stronę peryferium,
- **MISO** — dane w stronę kontrolera,
- **CS** — wybór konkretnego urządzenia.

SPI samo z siebie nie przesyła adresu urządzenia. To linia **CS** fizycznie wskazuje, kto ma uczestniczyć w transmisji.

SPI to dwa rejestry przesuwne pracujące jednocześnie

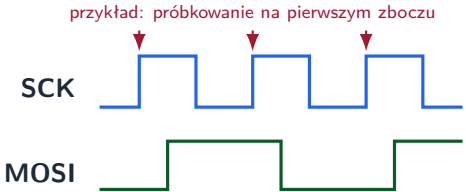


Przy każdym takcie SCK:

- controller wysuwa jeden bit na MOSI,
- peripheral wysuwa jeden bit na MISO,
- obie strony jednocześnie próbkują bit wejściowy.

Dlatego SPI jest naturalnie **full duplex**: podczas wysyłania bajtu zawsze równocześnie odbieramy bajt — nawet jeśli jedną stronę ignorujemy.

CPOL i CPHA: trzeba uzgodnić, kiedy dane są próbkowane



Dane muszą być stabilne w chwili próbkowania. Dlatego obie strony muszą używać **tego samego trybu SPI**.

CPOL – poziom spoczynkowy SCK:

- **0**: SCK w spoczynku jest LOW,
- **1**: SCK w spoczynku jest HIGH.

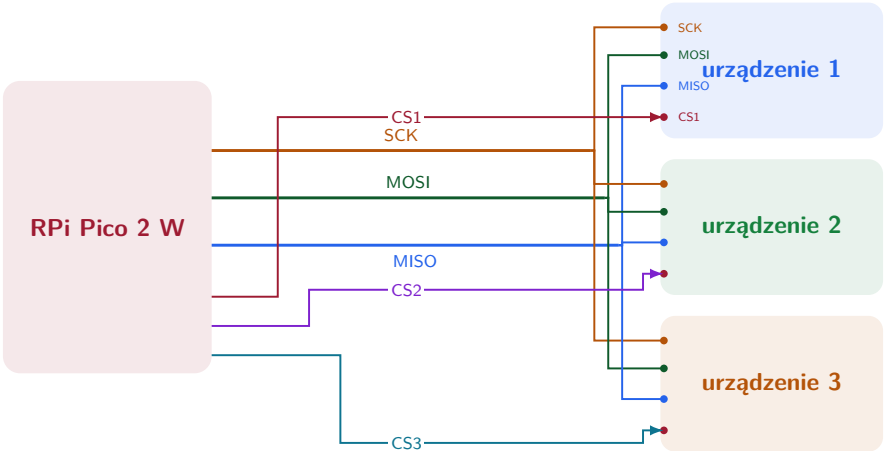
CPHA – na którym zboczu próbkujemy dane:

- **0**: na pierwszym zboczu okresu,
- **1**: na drugim zboczu okresu.

tryb	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Jak wybrać? Sprawdzamy datasheet peryferium. Np. **SPI mode 3** oznacza w MicroPythonie `polarity=1`, `phase=1`.

Kilka urządzeń SPI: linie danych wspólne, CS osobne



SCK, MOSI i MISO mogą być wspólne dla wielu urządzeń. **Aktywne jest tylko urządzenie, którego CS wybrano**. Każde dodatkowe urządzenie zwykle wymaga kolejnego GPIO na CS; SPI nie ma wspólnego adresowania takiego jak I^2C.

SPI w MicroPythonie: CS obsługujemy osobnym GPIO

```
from machine import Pin, SPI

spi = SPI(
    0,
    baudrate=1_000_000,
    polarity=0,
    phase=0,
    sck=Pin(6),
    mosi=Pin(7),
    miso=Pin(4)
)

cs = Pin(5, Pin.OUT, value=1)

tx = b'\x9F'
rx = bytearray(1)

cs.off()
spi.write_readinto(tx, rx)
cs.on()
```

- `baudrate` ustala częstotliwość SCK,
- `polarity`/`phase` wybierają tryb SPI,
- `write_readinto()` jednocześnie nadaje i odbiera,
- CS nie jest częścią obiektu `SPI` — najczęściej sterujemy nim jako zwykłym GPIO.

Sekwencję CS i znaczenie bajtów zawsze sprawdzamy w dokumentacji konkretnego układu.

Przykład: odczyt JEDEC ID pamięci SPI

```
# 0x9F = polecenie Read JEDEC ID
cs.off()
spi.write(b'\x9F')
jedec_id = spi.read(3, 0x00)
cs.on()

print(jedec_id.hex())
```

Czym jest JEDEC?

- JEDEC to organizacja tworząca standardy dla elektroniki półprzewodnikowej,
- w wielu pamięciach SPI komenda 0x9F zwraca ustandaryzowany identyfikator,
- typowe 3 bajty oznaczają: **producenta**, **typ pamięci** i **pojemność**.

Transakcja krok po kroku:

1. CS → LOW – wybieramy pamięć,
2. wysyłamy komendę 0x9F,
3. generujemy zegar dla 3 kolejnych bajtów,
4. pamięć zwraca identyfikator na MISO,
5. CS → HIGH – kończymy transakcję.

Po co są bajty wypełniające?

- SPI zawsze przesyła bity w obu kierunkach,
- aby odebrać dane z MISO, kontroler musi nadal generować SCK,
- równocześnie na MOSI wysyła bajty bez znaczenia, np. 0x00; w `spi.read(3, 0x00)` jest to bajt wypełniający,
- dodatkowe *dummy cycles*, jeśli są wymagane, wynikają z datasheet konkretnej komendy.

SPI w praktyce: dlaczego „szybciej” nie zawsze znaczy „lepiej”?

Na poziomie protokołu

- brak ACK/NACK w podstawowym SPI,
- brak standardowego adresowania,
- urządzenie może mieć własny format komend,
- często trzeba pilnować minimalnych czasów CS i opóźnień między poleceniami.

Na poziomie elektrycznym

- szybkie zbocza są bardziej wrażliwe na długie przewody,
- breadboard i przewody Dupont pogarszają jakość sygnałów,
- zbyt duży baud rate może dawać sporadyczne błędy,
- wszystkie urządzenia nadal potrzebują wspólnej masy.

W laboratorium zaczynamy od umiarkowanej częstotliwości, potwierdzamy poprawność transmisji, a dopiero później zwiększamy szybkość.

Pytanie kontrolne 2

Co w SPI wybiera konkretne urządzenie spośród kilku podłączonych do wspólnych SCK/MOSI/MISO?

A. adres w pierwszym bajcie

B. linia CS

C. bit ACK

D. rezystor pull-up

Pytanie kontrolne 2

Co w SPI wybiera konkretne urządzenie spośród kilku podłączonych do wspólnych SCK/MOSI/MISO?

A. adres w pierwszym bajcie

B. linia CS

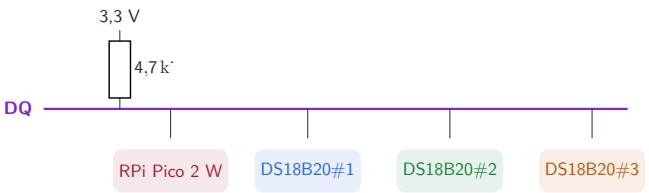
C. bit ACK

D. rezystor pull-up

1-Wire

Jedna linia danych, unikalne identyfikatory urządzeń

1-Wire: jedna linia danych współdzielona przez wiele urządzeń

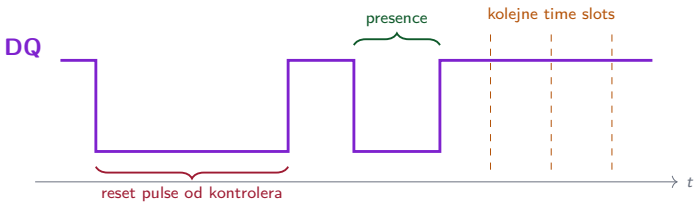


Jedna linia DQ przenosi dane w obie strony. Magistrala wykorzystuje strukturę open-drain i wspólny rezystor pull-up.

- brak osobnej linii zegara,
- czas bitu jest zakodowany w **długości impulsów**,
- każde urządzenie ma unikalny **64-bitowy kod ROM**,
- wiele czujników może pracować na tej samej linii,
- popularny przykład: DS18B20.

W MicroPythonie sterownik 1-Wire jest realizowany programowo; nie jest to osobny sprzętowy kontroler RP2350.

Reset, presence pulse i szczeliny czasowe

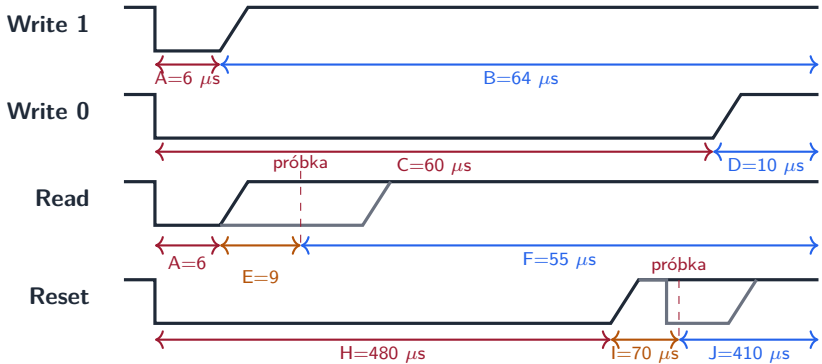


Komunikacja zaczyna się od **resetu**:

1. controller wymusza długi LOW,
2. puszcza linię,
3. obecne urządzenia odpowiadają impulsem **presence**,
4. dalsze bity przesyłane są w ściśle zdefiniowanych szczelinach czasowych.

To jest komunikacja asynchroniczna, ale z bardzo precyzyjnymi zależnościami czasowymi.

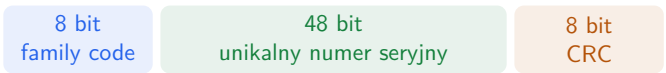
1-Wire: cztery podstawowe operacje i ich czasy



- **Czarny:** controller/master;
- **szary:** urządzenie 1-Wire; linia po zwolnieniu wraca do HIGH dzięki pull-up.
- Write 1 i Read zaczynają się krótkim LOW;
- Write 0 utrzymuje LOW prawie przez cały time slot.
- Dla Reset: $G = 0\,\mu$s, następnie długi LOW i sprawdzenie presence.

Czasy zalecane dla **standard speed**. Rysunek nie jest w skali.

Po co każdemu urządzeniu 64-bitowy kod ROM?



28-FF-64-1D-93-16-04-5C

Przykładowy zapis identyfikatora — każdy fizyczny egzemplarz ma własny kod.

Dzięki temu controller może:

- wykryć wszystkie urządzenia przez **ROM search**,
- wybrać jeden konkretny czujnik przez jego kod,
- wysłać polecenie do wszystkich urządzeń naraz, jeśli protokół to dopuszcza.

To funkcjonalnie rozwiązuje problem adresowania, ale inaczej niż 7-bitowy adres I^2C: identyfikator jest unikalny dla egzemplarza.

DS18B20 w MicroPythonie: jeden kod obsługuje wiele czujników

```
from machine import Pin
from time import sleep_ms
import onewire
import ds18x20

ow = onewire.OneWire(Pin(12))
ds = ds18x20.DS18X20(ow)

roms = ds.scan()
print(roms)

ds.convert_temp()
sleep_ms(750)

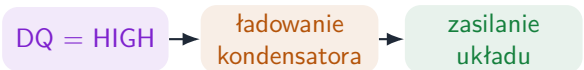
for rom in roms:
    print(rom, ds.read_temp(rom))
```

Kolejność jest ważna:

1. `scan()` znajduje kody ROM,
2. `convert_temp()` uruchamia pomiar,
3. czujnik potrzebuje czasu na konwersję,
4. dopiero potem odczytujemy wynik dla każdego ROM.

Na linii DQ potrzebny jest rezystor pull-up około **4,7 kΩ**.

1-Wire może także zasilać urządzenie z linii danych



W trybie *parasite power* DS18B20 pobiera energię z DQ, gdy linia jest HIGH. Zwykły pull-up 4,7 kΩ jest dobry do komunikacji, ale przy większym poborze prądu może nie wystarczyć.

Strong pull-up = czasowe, aktywne podciągnięcie DQ do zasilania przez element o małej rezystancji (np. tranzystor). Dostarcza większy prąd, np. podczas konwersji temperatury w trybie parasite power.

Na laboratorium najprościej:

- zasilać DS18B20 normalnie z 3,3 V,
- połączyć GND,
- DQ połączyć z GPIO,
- dodać 4,7 kΩ z DQ do 3,3 V.

„Jedna linia” w nazwie 1-Wire oznacza jedną linię **komunikacyjną**; masa nadal jest potrzebna.

Pytanie kontrolne 3

Co pozwala odróżnić kilka czujników DS18B20 pracujących na tej samej linii 1-Wire?

A. osobna linia CS

B. unikalny 64-bitowy kod ROM

C. inna częstotliwość zegara

D. osobny rezystor pull-up dla każdego

Pytanie kontrolne 3

Co pozwala odróżnić kilka czujników DS18B20 pracujących na tej samej linii 1-Wire?

A. osobna linia CS

B. unikalny 64-bitowy kod ROM

C. inna częstotliwość zegara

D. osobny rezystor pull-up dla każdego

UART, I²C, SPI i 1-Wire — co naprawdę je różni?

	UART	I²C	SPI	1-Wire
linie	TX, RX	SDA, SCL	SCK, MOSI, MISO + CS	DQ
zegar	brak	SCL od kontrolera	SCK od kontrolera	czas impulsów
wiele urządzeń	nie zalecane	adresy	osobne CS	64-bit ROM
elektryka	push-pull	open-drain + pull-up	zwykle push-pull	open-drain + pull-up
kierunek	TX/RX osobno	SDA w obie strony	full duplex	DQ w obie strony
przykład	GPS, drugi MCU	czujniki, RTC	pamięć, wyświetlacz	kilka DS18B20

Nie ma „najlepszej magistrali”. Jest interfejs **najlepiej dopasowany do konkretnego problemu**.

Jak wybrać interfejs? Cztery krótkie scenariusze

UART **Moduł GPS wysyła tekstowe ramki NMEA** Dwa urządzenia, ciągły strumień danych, brak potrzeby adresowania.

I2C **IMU + czujnik ciśnienia + RTC**. Kilka niezbyt szybkich urządzeń może współdzielić SDA/SCL; każde ma adres.

SPI **Wyświetlacz lub szybka pamięć.** Więcej przewodów, ale wysoka szybkość i prosty synchroniczny transfer.

1-Wire **Kilka DS18B20 w różnych miejscach.** Jedna linia danych i unikalne identyfikatory każdego czujnika.

Przed wyborem pytamy: **ile urządzeń, ile przewodów, jaka szybkość, jaka odległość i czy urządzenie w ogóle obsługuje dany interfejs?**