Od datasheetu do sterownika

Programowanie urządzeń peryferyjnych



Podstawy Systemów Mikroprocesorowych ▪ Wykład 5
dr inż. Dariusz Tefelski

Wykład 4 mówił: jak działa magistrala. Wykład 5: jak obsłużyć urządzenie.

Wykład 4

START/STOP, ACK,
CPOL/CPHA, CS,
open-drain, reset/presence

Jak przesyłamy bity?

Wykład 5

rejestry, maski, format danych,
status,
timeout, własny sterownik

Co znaczą przesłane bajty?

Po wykładzie chcemy umieć:

- znaleźć w datasheet rejestry i sekwencję inicjalizacji,
- złożyć/rozłożyć pola bitowe,
- odczytać dane wielobajtowe,
- napisać mały, czytelny sterownik.

Od datasheetu do bajtów

Rejestry, maski, format danych i stan urządzenia

Jak czytać datasheet bez czytania go od deski do deski?

Najpierw szukamy:

1. pinout i napięcia zasilania,
2. obsługiwany interfejs i adres/tryb,
3. tabelę **register map**,
4. rozdział „power-on/reset”,
5. kolejność inicjalizacji,
6. format danych pomiarowych.

Dopiero później:

- szczegółowe charakterystyki,
- parametry dokładności,
- tryby oszczędzania energii,
- przypadki brzegowe i opcje specjalne.

Datasheet to **mapa interfejsu urządzenia**, nie książka do czytania liniowo.

Mapa rejestrów: urządzenie udostępnia małe „pola pamięci”

adres	nazwa	typ	znaczenie
`0xD0`	ID	RO	identyfikacja układu
`0xF3`	STATUS	RO	stan pracy
`0xF4`	CTRL	R/W	konfiguracja pomiaru
`0xF7...`	DATA	RO	wynik pomiaru

RO — read only: czytamy, ale nie zapisujemy.

R/W — read/write: program może zmieniać konfigurację.

Adres rejestru **nie jest adresem urządzenia**. Najpierw wybieramy układ na magistrali, potem konkretny rejestr w środku układu.

Jeden rejestr często zawiera kilka niezależnych pól

Jeden bajt może sterować kilkoma funkcjami naraz.

- pojedyncze pole może mieć 1, 2, 3 lub więcej bitów,
- datasheet podaje pozycję bitów i znaczenie każdej wartości,
- sąsiednie pola mogą sterować zupełnie innymi funkcjami.

Dlatego potrzebujemy operacji bitowych:

- » / « — przesuwanie pól,
- & — wybieranie lub zerowanie wybranych bitów,
- | — ustawianie i łączenie bitów,
- ~ — odwrócenie maski.

Nie ustawiamy „rejestru na 5”. Ustawiamy **konkretne pole** w rejestrze, nie zmieniając pozostałych pól.

Maska bitowa: jak wyciągnąć pole z rejestru?




```
reg = 0b10110100

# bity 4..2
rate = (reg >> 2) & 0b111
```

Krok 1 — przesunięcie:

$$10110100 \gg 2 = 00101101$$

Krok 2 — maskowanie AND:

$$\begin{array}{r} 00101101 \\ \&\ 00000111 \\ \hline 00000101 \end{array}$$

Maska `00000111` pozostawia tylko trzy najmłodsze bity; wynik to `0b101` = 5.

Jak złożyć kilka pól w jeden bajt konfiguracji?




```
mode = 0b101
rate = 0b011
enable = 0b01

reg = (mode << 5) | (rate << 2) | enable
```

Każde pole najpierw trafia na swoją pozycję:

```
mode << 5 :  10100000
rate << 2 :  00001100
   enable :  00000001
 wynik OR :  10101101

10101101 = 0xAD
```

Jak ustawić bit na 1 lub 0 bez zmiany pozostałych?

ustaw bit 3 na 1

Tworzymy maskę z jedynką tylko na bicie 3 i wykonujemy OR:

```
  10100100
| 00001000
----------
  10101100
```

OR z zerem nie zmienia pozostałych bitów.

wyzeruj bit 3

Najpierw tworzymy maskę i ją negujemy:

```
maska :  00001000
~maska :  11110111
```

Dopiero potem wykonujemy AND:

```
  10101100
& 11110111
----------
  10100100
```

AND z jedynką pozostawia pozostałe bity bez zmian.

Dla bitu `n`: ustawienie — `reg | (1 << n)`, wyzerowanie — `reg & ~(1 << n)`.

Zmiana jednego pola bez niszczenia pozostałych bitów

```
# chcemy zmienic tylko bity 4..2
MASK = 0b111 << 2

reg = reg & ~MASK # wyzeruj pole
reg = reg | (rate << 2)
# wpisz nowa wartosc
```

To jest wzorzec **read-modify-write**: odczytaj rejestr, zmień wskazane pole, zapisz całość z powrotem.

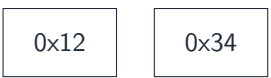
Przykład: reg=11010101, nowe rate=010.

```
                MASK :  00011100
               ~MASK :  11100011
11010101 & 11100011 :  11000001
         rate << 2 :  00001000
11000001 | 00001000 :  11001001
```

Zmieniły się wyłącznie bity 4..2; wszystkie pozostałe zachowały wcześniejszą wartość.

Wiele bajtów: w jakiej kolejności tworzą liczbę?

big-endian



starszy bajt pierwszy

0x1234

little-endian



młodszy bajt pierwszy

0x1234

Kolejności **nie zgadujemy**. Zawsze wynika z dokumentacji konkretnego urządzenia.

`struct.unpack()`: mniej ręcznego składania bajtów

```
import struct

raw = b'\x12\x34'

value = struct.unpack('>H', raw)[0]
# > big-endian
# H unsigned 16-bit
```

Przykładowe kody formatu:

- `B` — unsigned 8-bit,
- `b` — signed 8-bit,
- `H`/`h` — unsigned/signed 16-bit,
- `>`/`<` — big-/little-endian.

Signed i unsigned: ten sam wzorzec bitów może znaczyć coś innego



ten sam bajt: `0b11111100`

unsigned 8-bit	**signed 8-bit, U2**
252	-4

Szczególnie ważne dla temperatur, przyspieszeń i współczynników kalibracyjnych.

RAW, skalowanie i jednostki — trzy różne rzeczy



Dla jednego czujnika wystarczy podzielić przez stałą. Inny wymaga współczynników kalibracyjnych zapisanych w pamięci układu. Datasheet mówi, **jak przejść od bitów do wielkości fizycznej**.

Rejestr STATUS: polling gotowości przez maskę bitową

Przykład pola statusowego:

$$\text{STATUS} = \texttt{0b00001000}$$

bit 3: `measuring`=1 $\Rightarrow$ pomiar trwa

`MEASURING_MASK = 1 « 3` to **maska wybierająca bit 3**, czyli `0b00001000`.

```
REG_STATUS = 0xF3
MEASURING_MASK = 1 << 3

while True:
    status = read_reg(REG_STATUS)
    if (status & MEASURING_MASK) == 0:
        break
    sleep_ms(2)
```

Zamiast zakładać stały czas, program **pyta urządzenie o stan** i kończy czekanie, gdy bit `measuring` wróci do 0.

Każde oczekiwanie na sprzęt powinno mieć timeout

```
from time import ticks_ms, ticks_diff

start = ticks_ms()
while read_reg(REG_STATUS) & MEASURING_MASK:
    if ticks_diff(ticks_ms(), start) > 100:
        raise OSError('measurement timeout')
```

Bez timeoutu uszkodzony czujnik, zerwana magistrala lub błąd konfiguracji może zatrzymać cały program na zawsze.

Pytanie kontrolne 1

Dlaczego funkcja sterownika powinna mieć timeout podczas oczekiwania na gotowość urządzenia?

A. Aby zmienić endianowość danych

B. Aby zwiększyć częstotliwość magistrali

C. Aby program nie utknął na zawsze, gdy urządzenie nie zakończy operacji poprawnie

D. Tylko po to, aby skrócić kod

Pytanie kontrolne 1

Dlaczego funkcja sterownika powinna mieć timeout podczas oczekiwania na gotowość urządzenia?

A. Aby zmienić endianowość danych

B. Aby zwiększyć częstotliwość magistrali

C. Aby program nie utknął na zawsze, gdy urządzenie nie zakończy operacji poprawnie

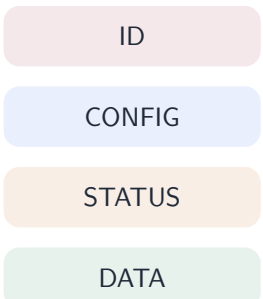
D. Tylko po to, aby skrócić kod

Case study: BME280

Od tabeli rejestrów do funkcji read()

Dlaczego BME280 jest dobrym przykładem sterownika?

- ma czytelną mapę rejestrów,
- posiada rejestry ID, statusu, konfiguracji i danych,
- dane są wielobajtowe,
- wymaga współczynników kalibracyjnych,
- może działać przez I^2C albo SPI.



Jak działa BME280?

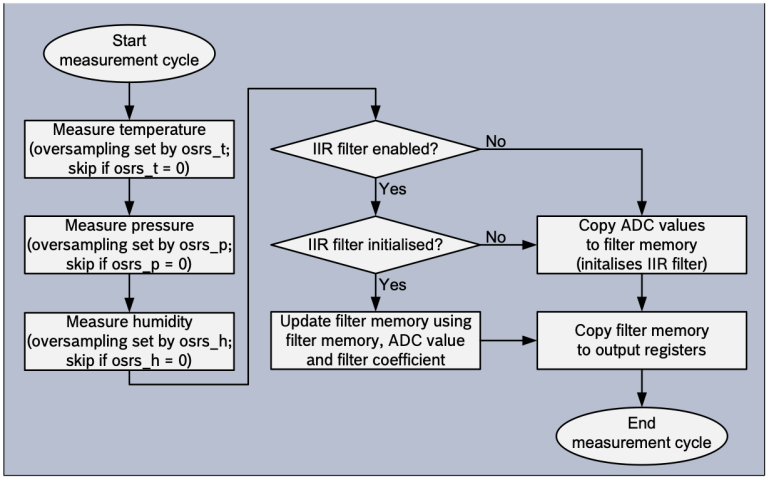


Figure 6: BME280 measurement cycle

Zaczynamy od „czy rozmawiam z właściwym układem?”

adres I^2C ► rejestr 0xD0 ► oczekujemy 0x60

Rejestr identyfikacyjny to szybki test: połączenia, adres i podstawowa komunikacja działają, zanim zaczniemy analizować pomiary.

```
from machine import Pin, I2C

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

ADDRESS = 0x76
REG_ID = 0xD0

chip_id = i2c.readfrom_mem(ADDRESS, REG_ID, 1)[0]
print(hex(chip_id)) # 0x60
```

Inicjalizacja sterownika powinna weryfikować układ

1. utwórz obiekt magistrali,
2. odczytaj ID,
3. jeśli ID jest inne — przerwij inicjalizację,
4. wykonaj reset,
5. poczekaj na zakończenie kopiowania danych kalibracyjnych,
6. zapisz konfigurację.

Sterownik nie powinien zakładać, że „na pewno podłączono właściwy moduł". Powinien to **sprawdzić**.



To dobry przykład, dlaczego po komendzie nie zawsze wolno natychmiast wysłać następną.

BME280: konfiguracja jest rozbita na kilka rejestrów

Register Name	Address	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reset state
hum_lsb	0xFE	hum_lsb<7:0>								0x00
hum_msb	0xFD	hum_msb<7:0>								0x80
temp_xlsb	0xFC	temp_xlsb<7:4>				0	0	0	0	0x00
temp_lsb	0xFB	temp_lsb<7:0>								0x00
temp_msb	0xFA	temp_msb<7:0>								0x80
press_xlsb	0xF9	press_xlsb<7:4>				0	0	0	0	0x00
press_lsb	0xF8	press_lsb<7:0>								0x00
press_msb	0xF7	press_msb<7:0>								0x80
config	0xF5	t_sb[2:0]			filter[2:0]				spi3w_en[0]	0x00
ctrl_meas	0xF4	osrs_t[2:0]			osrs_p[2:0]			mode[1:0]		0x00
status	0xF3					measuring[0]			im_update[0]	0x00
ctrl_hum	0xF2						osrs_h[2:0]			0x00
calib26..calib41	0xE1...0xF0	calibration data								individual
reset	0xE0	reset[7:0]								0x00
id	0xD0	chip_id[7:0]								0x60
calib00..calib25	0x88...0xA1	calibration data								individual

Registers:	Reserved registers	Calibration data	Control registers	Data registers	Status registers	Chip ID	Reset
Type:	do not change	read only	read / write	read only	read only	read only	write only

adres	rejestr	co ustawiamy
0xF2	`ctrl_hum`	oversampling wilgotności
0xF4	`ctrl_meas`	oversampling T/P + tryb pracy
0xF5	`config`	standby, filtr, 3-wire SPI

Oversampling — układ wykonuje więcej wewnętrznych próbek dla jednego wyniku. Większa wartość zmniejsza szum i zwiększa efektywną rozdzielczość, ale **wydłuża pomiar** i zwiększa pobór energii.

Standby — przerwa pomiędzy kolejnymi automatycznymi cyklami w *normal mode*. W *forced mode* po pojedynczym pomiarze układ wraca do sleep, więc standby nie steruje tym oczekiwaniem.

Zmiana `ctrl_hum` zaczyna obowiązywać po kolejnym zapisie `ctrl_meas` — kolejność zapisów również może być częścią obsługi urządzenia.

Budujemy wartość `ctrl_meas` z nazwanych pól

Register 0xF4 *“ctrl_meas”*	Name	Description
Bit 7, 6, 5	osrs_t[2:0]	Controls oversampling of temperature data. See Table 24 for settings and chapter 3.4.3 for details.
Bit 4, 3, 2	osrs_p[2:0]	Controls oversampling of pressure data. See Table 23 for settings and chapter 3.4.2 for details.
Bit 1, 0	mode[1:0]	Controls the sensor mode of the device. See Table 25 for settings and chapter 3.3 for details.

```
OSRS_T_X1 = 0b001
OSRS_P_X1 = 0b001
MODE_FORCED = 0b01

ctrl_meas = (OSRS_T_X1 << 5) | \
       (OSRS_P_X1 << 2) | \
       MODE_FORCED
```

Table 24: register settings *osrs_t*

osrs_t[2:0]	Temperature oversampling
000	Skipped (output set to 0x80000)
001	oversampling ×1
010	oversampling ×2
011	oversampling ×4
100	oversampling ×8
101, others	oversampling ×16

Budujemy wartość `ctrl_meas` z nazwanych pól

Register 0xF4 *“ctrl_meas”*	Name	Description
Bit 7, 6, 5	osrs_t[2:0]	Controls oversampling of temperature data. See Table 24 for settings and chapter 3.4.3 for details.
Bit 4, 3, 2	osrs_p[2:0]	Controls oversampling of pressure data. See Table 23 for settings and chapter 3.4.2 for details.
Bit 1, 0	mode[1:0]	Controls the sensor mode of the device. See Table 25 for settings and chapter 3.3 for details.

```
OSRS_T_X1 = 0b001
OSRS_P_X1 = 0b001
MODE_FORCED = 0b01

ctrl_meas = (OSRS_T_X1 << 5) | \
    (OSRS_P_X1 << 2) | \
    MODE_FORCED
```

Table 23: register settings *osrs_p*

osrs_p[2:0]	Pressure oversampling
000	Skipped (output set to 0x80000)
001	oversampling ×1
010	oversampling ×2
011	oversampling ×4
100	oversampling ×8
101, others	oversampling ×16

Forced mode: jeden pomiar na żądanie

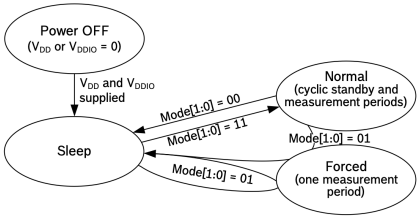


Figure 3: Sensor mode transition diagram

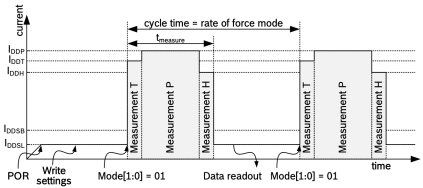


Figure 4: Forced mode timing diagram

- zapis `mode=01` uruchamia jeden cykl pomiarowy,
- w czasie konwersji bit `measuring`=1,
- po zakończeniu wynik trafia do rejestrów danych,
- BME280 automatycznie wraca do *sleep mode*.

zapis `ctrl_meas` → MODE=forced → measuring=1 → measuring=0 → odczyt danych

Budujemy wartość `ctrl_meas` z nazwanych pól

Register 0xF4 *“ctrl_meas”*	Name	Description
Bit 7, 6, 5	osrs_t[2:0]	Controls oversampling of temperature data. See Table 24 for settings and chapter 3.4.3 for details.
Bit 4, 3, 2	osrs_p[2:0]	Controls oversampling of pressure data. See Table 23 for settings and chapter 3.4.2 for details.
Bit 1, 0	mode[1:0]	Controls the sensor mode of the device. See Table 25 for settings and chapter 3.3 for details.

```
OSRS_T_X1 = 0b001
OSRS_P_X1 = 0b001
MODE_FORCED = 0b01

ctrl_meas = (OSRS_T_X1 << 5) | \
    (OSRS_P_X1 << 2) | \
    MODE_FORCED
```

Table 25: register settings *mode*

***mode*[1:0]**	**Mode**
00	Sleep mode
01 and 10	Forced mode
11	Normal mode

Funkcja `wait_ready()` z timeoutem

```
REG_STATUS = 0xF3
MEASURING = 1 << 3 # bit 3, czyli 0x08

def wait_ready(timeout_ms=100):
    start = ticks_ms()
    while read_u8(REG_STATUS) & MEASURING:
        if ticks_diff(ticks_ms(), start) > timeout_ms:
            raise OSError('BME280 timeout')
        sleep_ms(2)
```

`REG_STATUS` wskazuje rejestr `0xF3`, a `MEASURING` jest maską jego bitu 3. Gdy ten bit wróci do 0, konwersja jest zakończona.

Register 0xF3 *"status"*	Name	Description
Bit 3	measuring[0]	Automatically set to '1' whenever a conversion is running and back to '0' when the results have been transferred to the data registers.
Bit 0	im_update[0]	Automatically set to '1' when the NVM data are being copied to image registers and back to '0' when the copying is done. The data are copied at power-on-reset and before every conversion.

Burst read: jeden transfer zamiast wielu pojedynczych odczytów



Jeden odczyt blokowy zmniejsza narzut magistrali i daje spójny zestaw danych z jednego cyklu pomiarowego.

20-bitowa wartość: trzy bajty, ale tylko część ostatniego

```
raw = i2c.readfrom_mem(ADDRESS, 0xF7, 8)

adc_p = (raw[0] << 12) | (raw[1] << 4) | (raw[2] >> 4)
```

Ostatni bajt nie zawsze jest używany w całości. To kolejny powód, by patrzeć na diagram formatu danych w datasheet.

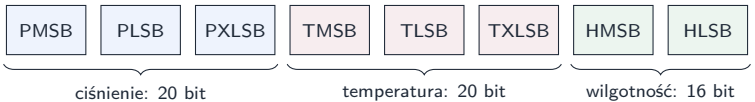
Table 29: Register 0xF7 ... 0xF9 *“press”*

Register 0xF7...0xF9 *“press”*	**Name**	**Description**
0xF7	press_msb[7:0]	Contains the MSB part up[19:12] of the raw pressure measurement output data.
0xF8	press_lsb[7:0]	Contains the LSB part up[11:4] of the raw pressure measurement output data.
0xF9 (bit 7, 6, 5, 4)	press_xlsb[3:0]	Contains the XLSB part up[3:0] of the raw pressure measurement output data. Contents depend on temperature resolution.

Jeden burst: składamy ciśnienie, temperaturę i wilgotność

```
raw = i2c.readfrom_mem(ADDRESS, 0xF7, 8)

adc_p = ((raw[0] << 12) | (raw[1] << 4) | (raw[2] >> 4))
adc_t = ((raw[3] << 12) | (raw[4] << 4) | (raw[5] >> 4))
adc_h = (raw[6] << 8) | raw[7]
```



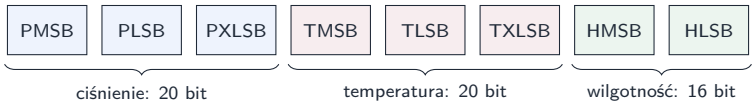
To nadal są wartości **RAW**. Dopiero kompensacja z użyciem współczynników zapisanych w BME280 daje °C, Pa i %RH.

Register 0xFA...0xFC *"temp"*	Name	Description
0xFA	temp_msb[7:0]	Contains the MSB part ut[19:12] of the raw temperature measurement output data.
0xFB	temp_lsb[7:0]	Contains the LSB part ut[11:4] of the raw temperature measurement output data.
0xFC (bit 7, 6, 5, 4)	temp_xlsb[3:0]	Contains the XLSB part ut[3:0] of the raw temperature measurement output data. Contents depend on pressure resolution.

Jeden burst: składamy ciśnienie, temperaturę i wilgotność

```
raw = i2c.readfrom_mem(ADDRESS, 0xF7, 8)

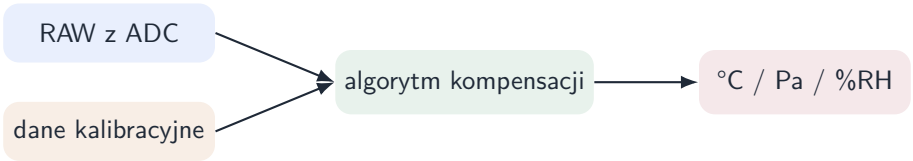
adc_p = ((raw[0] << 12) | (raw[1] << 4) | (raw[2] >> 4))
adc_t = ((raw[3] << 12) | (raw[4] << 4) | (raw[5] >> 4))
adc_h = (raw[6] << 8) | raw[7]
```



To nadal są wartości **RAW**. Dopiero kompensacja z użyciem współczynników zapisanych w BME280 daje °C, Pa i %RH.

Register 0xFD...0xFE *"hum"*	Name	Description
0xFD	hum_msb[7:0]	Contains the MSB part uh[15:8] of the raw humidity measurement output data.
0xFE	temp_lsb[7:0]	Contains the LSB part uh[7:0] of the raw humidity measurement output data.

Kalibracja: współczynniki czytamy raz podczas inicjalizacji




```
import struct

# wspolczynniki temperatury: 0x88..0x8D
cal = i2c.readfrom_mem(ADDRESS, 0x88, 6)
dig_T1, dig_T2, dig_T3 = \\
    struct.unpack('<Hhh', cal)
```

`<` oznacza little-endian, `H` — unsigned 16-bit, a `h` — signed 16-bit.

BME280 przechowuje współczynniki indywidualne dla egzemplarza, dlatego sterownik odczytuje je **raz podczas inicjalizacji** i wykorzystuje przy każdym pomiarze.

Pytanie kontrolne 2

Po co sterownik BME280 odczytuje współczynniki kalibracyjne?

A. Aby ustawić częstotliwość I^2C

B. Aby zmienić adres 0x76 na 0x77

C. Tylko do sprawdzenia CRC

D. Ponieważ surowe wyniki wymagają kompensacji zależnej od konkretnego egzemplarza

Pytanie kontrolne 2

Po co sterownik BME280 odczytuje współczynniki kalibracyjne?

A. Aby ustawić częstotliwość I^2C

B. Aby zmienić adres 0x76 na 0x77

C. Tylko do sprawdzenia CRC

D. Ponieważ surowe wyniki wymagają kompensacji zależnej od konkretnego egzemplarza

Pytanie kontrolne 3

Co jest najlepszym pierwszym testem po podłączeniu urządzenia rejestrowego?

A. Odczytać rejestr identyfikacyjny i porównać z oczekiwaną wartością

B. Od razu przeliczyć wynik na jednostki fizyczne

C. Zwiększyć częstotliwość magistrali do maksimum

D. Zmienić wszystkie rejestry konfiguracyjne

Pytanie kontrolne 3

Co jest najlepszym pierwszym testem po podłączeniu urządzenia rejestrowego?

A. Odczytać rejestr identyfikacyjny i porównać z oczekiwaną wartością

B. Od razu przeliczyć wynik na jednostki fizyczne

C. Zwiększyć częstotliwość magistrali do maksimum

D. Zmienić wszystkie rejestry konfiguracyjne

Drugi model: DS18B20

Urządzenie bez klasycznej mapy rejestrów

1-Wire pokazuje, że sterownik nie zawsze opiera się na rejestrach

W DS18B20 mamy:

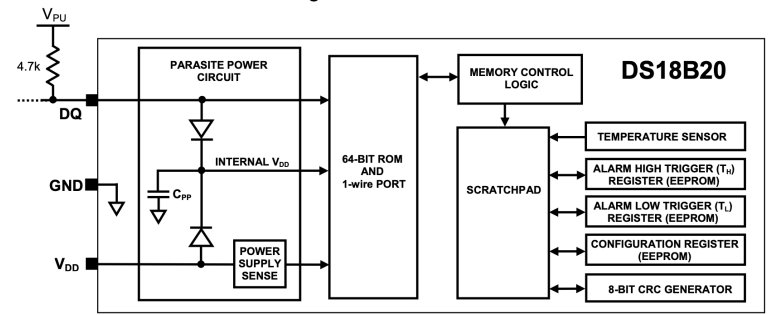
- komendy ROM do wyboru urządzenia,
- komendy funkcjonalne, np. Convert T,
- scratchpad z wynikiem i konfiguracją,
- CRC.

RESET + ROM

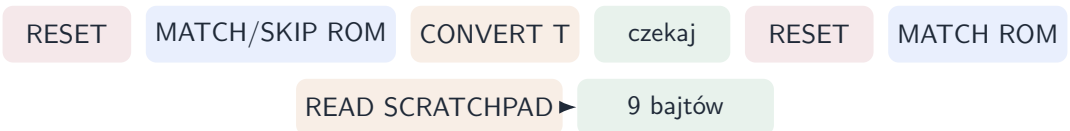
COMMAND

SCRATCHPAD

DS18B20 BLOCK DIAGRAM Figure 1



Pełny pomiar DS18B20 to sekwencja poleceń



Polecenia z dokumentacji

DS18B20 FUNCTION COMMAND SET Table 4

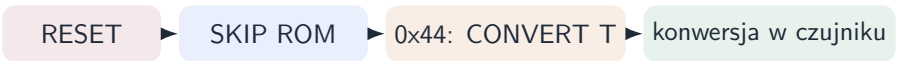
Command	Description	Protocol	1-Wire Bus Activity After Command is Issued	Notes
TEMPERATURE CONVERSION COMMANDS				
Convert T	Initiates temperature conversion.	44h	DS18B20 transmits conversion status to master (not applicable for parasite-powered DS18B20s).	1
MEMORY COMMANDS				
Read Scratchpad	Reads the entire scratchpad including the CRC byte.	BEh	DS18B20 transmits up to 9 data bytes to master.	2
Write Scratchpad	Writes data into scratchpad bytes 2, 3, and 4 (T_H, T_L, and configuration registers).	4Eh	Master transmits 3 data bytes to DS18B20.	3
Copy Scratchpad	Copies T_H, T_L, and configuration register data from the scratchpad to EEPROM.	48h	None	1
Recall E^2	Recalls T_H, T_L, and configuration register data from EEPROM to the scratchpad.	B8h	DS18B20 transmits recall status to master.	
Read Power Supply	Signals DS18B20 power supply mode to the master.	B4h	DS18B20 transmits supply status to master.	

Co naprawdę robi `convert_temp()`?

```
CONVERT_T = 0x44

def convert_temp_low_level(ow):
    ow.reset()
    ow.writebyte(ow.SKIP_ROM)
    ow.writebyte(CONVERT_T)
```

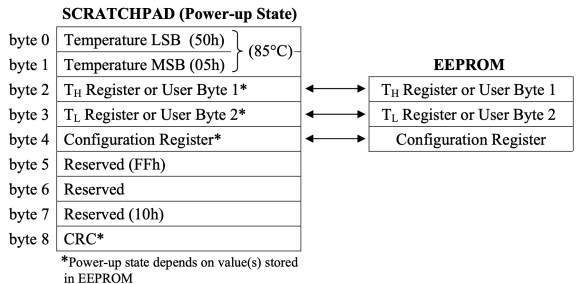
1. `reset()` rozpoczyna nową transakcję i sprawdza presence,
2. `SKIP_ROM` wybiera wszystkie czujniki na magistrali,
3. `0x44` = **Convert T** — rozpoczyna konwersję temperatury,
4. funkcja **nie odczytuje jeszcze wyniku** i sama nie musi czekać do końca konwersji.



Po wysłaniu `Convert T` trzeba poczekać na zakończenie konwersji — wymagany czas zależy od ustawionej rozdzielczości. Dopiero potem rozpoczynamy nową transakcję: wybieramy konkretny ROM i wysyłamy `0xBE` (Read Scratchpad).

Scratchpad: wynik, progi, konfiguracja i CRC w jednym bloku

DS18B20 MEMORY MAP Figure 7



bajt	nazwa	rola
0	TEMP LSB	młodsza część temperatury
1	TEMP MSB	starsza część temperatury
2–3	TH / TL	progi alarmowe
4	CONFIG	rozdzielczość
5–7	reserved	wartości zdefiniowane przez układ
8	CRC	kontrola całego scratchpadu

Temperatura DS18B20: dlaczego dzielimy wartość RAW przez 16?

```
import struct

raw_temp = struct.unpack(
    '<h', scratchpad[0:2]
)[0]

temperature = raw_temp / 16.0
```

'<h' oznacza: little-endian + signed 16-bit.

W trybie 12-bitowym cztery najmłodsze bity opisują część ułamkową temperatury. Waga LSB wynosi:

$$2^{-4}\ ^\circ\mathrm{C} = \frac{1}{16}\ ^\circ\mathrm{C} = 0.0625\ ^\circ\mathrm{C}$$

Dlatego liczba całkowita RAW jest temperaturą pomnożoną przez 16.

Przykład: `0x0191 = 401`, więc

$$401/16 = 25.0625\ ^\circ\mathrm{C}.$$

TEMPERATURE REGISTER FORMAT Figure 2

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
LS Byte	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
	bit 15	**bit 14**	**bit 13**	**bit 12**	**bit 11**	**bit 10**	**bit 9**	**bit 8**
MS Byte	S	S	S	S	S	2^6	2^5	2^4

Wiele czujników: jedna komenda może rozpocząć konwersję równolegle



Potem kolejne czujniki wybieramy przez ich ROM i odczytujemy scratchpad. To lepsze niż uruchamianie pełnej konwersji osobno dla każdego sensora.

Pytanie kontrolne 4

Dlaczego `struct.unpack('<h', ...)` pasuje do temperatury DS18B20?

A. Bo dane są big-endian i unsigned

B. Bo wynik ma zawsze 8 bitów

C. Bo temperatura jest 16-bitowa, signed i przesyłana młodszym bajtem jako pierwszym

D. Bo znak < oznacza jednostkę °C

Pytanie kontrolne 4

Dlaczego `struct.unpack('<h', ...)` pasuje do temperatury DS18B20?

A. Bo dane są big-endian i unsigned

B. Bo wynik ma zawsze 8 bitów

C. Bo temperatura jest 16-bitowa, signed i przesyłana młodszym bajtem jako pierwszym

D. Bo znak < oznacza jednostkę °C

Mały sterownik: prosty interfejs, szczegóły w środku

```
class BME280:
    REG_STATUS = 0xF3
    MEASURING = 1 << 3

    def __init__(self, i2c, address=0x76):
        self.i2c = i2c
        self.address = address
        self._check_id()
        self._read_calibration()

    def read(self):
        self._start_measurement()
        self._wait_ready()
        return self._read_values()
```

Sterownik:

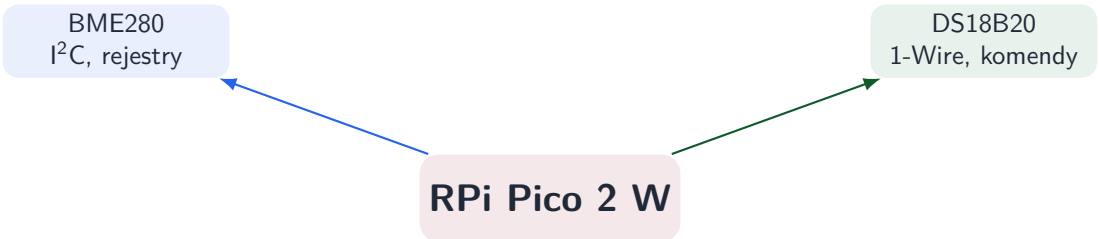
- przechowuje **odwołanie do obiektu magistrali** i adres urządzenia,
- zna rejestry, maski i komendy,
- podczas inicjalizacji może sprawdzić ID i odczytać dane kalibracyjne,
- pilnuje timeoutów/CRC i zgłasza czytelne błędy,
- zwraca aplikacji gotowe wartości, np. °C i %RH.

Aplikacja wywołuje `env.read()` — nie musi wiedzieć, który bit w `0xF4` uruchamia pomiar.

Sterownik a aplikacja: kto za co odpowiada?

sterownik urządzenia	**aplikacja**
zna rejestry, komendy i format danych	wie, po co i kiedy potrzebny jest pomiar
obsługuje I^2C/SPI/1-Wire	decyduje, kiedy wywołać pomiar
sprawdza timeout / CRC	reaguje na błąd w kontekście całego systemu
przelicza RAW na jednostki fizyczne	wyświetla, zapisuje lub wysyła wynik

Przykład systemu: dwa czujniki, dwa różne modele obsługi



Aplikacja powinna widzieć dwa proste interfejsy: `env.read()` i `temps.read()`. Nie musi znać adresów rejestrów BME280, kodów komend DS18B20 ani sposobu składania bajtów.

Dwa urządzenia mają zupełnie inny model komunikacji, ale dobry sterownik ukrywa te różnice przed aplikacją.

Co powinniśmy umieć po tym wykładzie?

- znaleźć w datasheet mapę rejestrów i sekwencję inicjalizacji,
- odczytywać i modyfikować pola bitowe,
- rozróżniać endianowość oraz signed/unsigned,
- używać `struct.unpack()`,
- odpytywać status urządzenia z timeoutem,
- rozumieć różnicę między RAW a wartością fizyczną,
- rozdzielać warstwę transportu od logiki urządzenia,
- zaprojektować prosty interfejs klasy sterownika.

Najważniejsza myśl

Magistrala przesyła bajty.

Datasheet mówi, co te bajty znaczą.

Sterownik zamienia tę wiedzę w prosty interfejs dla aplikacji.

Źródła ilustracji: Bosch Sensortec, *BME280 Data Sheet*:
`https://cdn.sparkfun.com/assets/learn_tutorials/4/1/9/BST-BME280_DS001-10.pdf`
Analog Devices/Maxim, *DS18B20 Data Sheet*: `https://labe.engined.eu/data/_uploaded/file/psm/materialy/Noty_katalogowe/DS18B20.pdf`
Pozostałe źródła techniczne: dokumentacja MicroPython.