

# POLITECHNIKA BYDGOSKA

Wydział Telekomunikacji,  
Informatyki i Elektrotechniki

## Projekt: Mikroprocesory

---

### Obsługa czujnika BH1750 przy użyciu STM32F446RE Dokumentacja projektu

KIERUNEK	INFORMATYKA STOSOWANA
TRYB	NIESTACJONARNIE
ROK AKADEMICKI	2025/2026 ZIMA
AUTOR	SZYMON BANASZEK

## Spis Treści

### 1. Specyfikacja projektu

### 2. Protokół komunikacyjny

1. Ramka
2. Kontrola poprawności
3. Komendy
4. Obsługa błędów
5. Implementacja w kodzie:
  - a. Definicje typów
  - b. Definicje
  - c. Zmienne
  - d. Funkcje USART
  - e. Odczyt ramki
  - f. Walidacja ramki
  - g. Obsługa ramki
  - h. Funkcje pomocnicze

### 3. Parametry Komunikacji

1. Konfiguracja USART2
2. Konfiguracja I2C
3. Konfiguracja TIM3

### 4. Opis czujnika BH1750

## 1. Specyfikacja Projektu

1. Oprogramować komunikację mikroprocesora z PC, poprzez interfejs asynchroniczny z wykorzystaniem przerwań oraz buforów kołowych.
2. Zaprojektować i zaimplementować protokół komunikacyjny pozwalający na adresowanie ramek, przekazywanie dowolnych danych oraz weryfikacje poprawności danych z uwzględnieniem ich kolejności.
3. Obsługa czujnika BH1750 poprzez I2C przy użyciu funkcji nieblokujących wspieranych przerwaniami. Dane zapisywać co zadany interwał czasy zdefiniowany w milisekundach uint32 i zapisywać do bufora cyklicznego o pojemności min 1000 wpisów umozliwić odczyt danych bieżących i archiwalnych.

## 2. Protokół komunikacyjny

1. Ramka
2. Kontrola poprawności

Każda ramka kończy się sumą kontrolną liczoną z Nadawcy, Odbiorcy, długości danych oraz danymi liczona z CRC-8.

3. Komendy

Kody Wysyłek:

- 10 – START
- 11 – STOP
- 12 – DOWNLOAD
- 13 – VIEW (+ xxzz)
- 14 – SET\_INTERVAL (+ xxxx)
- 15 – GET\_INTERVAL
- 16 – SET\_MODE (+ x)
- 17 – GET\_MODE

Kody Odbioru – Statusy :

- 00 – OK
- 01 – ERR\_PARAM – nieprawidłowe parametry
- 02 – ERR\_RANGE – liczba wpisana poza zakres
- 03 – ERR\_NO\_DATA – brak danych
- 04 – ERR\_I2C – błąd przy komunikacji I2C

1. Start pomiaru (10)
- Urządzenie zaczyna cyklicznie odczytywać dane z czujnika BH1750 w określonych odstępach czasu.
- Bufor ten działa niezależnie od komend pobierających – **pomiar odbywa się automatycznie po ST.**

### *Wysyłka*

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
----------------	---------	----------	----	----------------	------	----------------	--------------

&	PC_	STM	01	002	10	80	*
---	-----	-----	----	-----	----	----	---

*Odbiór*

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	STM	PC_	01	002	00	FC	*

## 2. Stop pomiaru (11)

- Zatrzymuje cykliczne pobieranie danych.
- Bufor nadal przechowuje wcześniej zmierzone dane.

*Wysyłka*

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	PC_	STM	02	002	11	FC	*

*Odbiór*

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	STM	PC_	02	002	00	87	*

## 3. Pobierz wpis (12)

- Pobiera najnowszy wpis z buforu.

*Wysyłka*

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	PC_	STM	03	002	12	DC	*

*Odbiór*

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	STM	PC_	03	003	100	DE	*

## 4. Podgląd wpisów (13xxzz)

- Szybki podgląd ostatnich N wpisów (bieżących) i opcjonalnie starszych wpisów (archiwum).
- Domyślnie wysyła ostatnie pomiary; opcjonalny parametr w polu Dane umożliwia wybranie starszych wpisów.
- xx – ID wpisu początkowego
- zz – ilość wpisów od początkowego ma wyświetlić

*Wysyłka*

- 13 = kod komendy VIEW
- 00 = ID początkowego wpisu (najnowsze pomiary)

- 02 = pobierz 2 pomiary

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	PC_	STM	04	006	130002	02	*

#### Odbiór

Każda ramka zawiera:

- ID początkowy ramki (2 cyfry)
- Ciąg wartości lux (każda po 4 cyfry)

Format DANE: xxLLLLLLL...

- xx = ID wpisu początkowego tej paczki
- LLLL = kolejne 4-cyfrowe wartości lux

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	STM	PC_	04	010	0050003500	A9	*

#### 5. Ustawienie interwału (14xxxx)

- Umożliwia **ustawienie interwału pomiarowego w milisekundach**.
- xxxx w polu Dane to **wartość interwału pomiarowego w milisekundach**.
- **Format:** 4 znaki w systemie dziesiętnym od 0 do 9 z dopełnieniem 0 do długości 4 cyfr , np. 1000 → 1000 ms (1 sekunda).
- Interwał wpływa na częstotliwość cyklicznego odczytu z BH1750.

#### Wysyłka

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	PC_	STM	05	006	145000	03	*

#### Odbiór

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	STM	PC_	05	002	00	58	*

#### 6. Odczyt interwału (15)

- Pobiera aktualnie ustawiony interwał pomiarowy.
- Dane zwracane w polu Dane po przesłaniu ramki.

#### Wysyłka

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	PC_	STM	06	002	15	44	*

*Odbiór*

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	STM	PC_	06	004	1000	A5	*

## 7. Ustawienie rozdzielczości (16x)

- x – to cyfra w systemie dziesiętnym
- 161 : Ciągły wysokiej rozdzielczości (1lx)
- 162 : Ciągły wysokiej rozdzielczości 2 (0,5lx)
- 163 : Ciągły niskiej rozdzielczości (4lx)
- 164: Ręczny wysokiej rozdzielczości (1lx)
- 165: Ręczny wysokiej rozdzielczości 2 (0,5lx)
- 166: Ręczny niskiej rozdzielczości (4lx)

*Wysyłka*

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	PC_	STM	07	003	161	BA	*

*Odbiór*

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	STM	PC_	07	002	00	0A	*

## 8. Pobranie rozdzielczości (17)

- Pobiera aktualnie ustawiony tryb rozdzielczości.

*Wysyłka*

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	PC_	STM	08	002	17	F3	*

*Odbiór*

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	STM	PC_	08	001	1	57	*

## 4. Obsługa błędów

*Błędy strukturalne ramki*

Gdy ramka jest niepoprawna strukturalnie, urządzenie nie może zbudować prawidłowej odpowiedzi w formacie protokołu, więc wysyła surowy tekst błędu.

**Ramka zbyt krótka**

- Długość ramki jest mniejsza niż 13
- Ramka odrzucona: **ERR: TOO SHORT**

#### Nieprawidłowe znaki w polach Nadawca/Odbiorca

- Znaki spoza zakresu ASCII (0x21-0x7E) lub znaki & i \*
- Ramka odrzucona: **ERR: INVALID ADDR**

#### Długość ID w złym formacie

- Pole ID zawiera znaki inne niż cyfry od 0 do 9
- Ramka odrzucona: **ERR: INVALID ID**

#### Pole LEN w złym formacie

- Pole LEN zawiera znaki inne niż cyfry od 0 do 9
- Ramka odrzucona: **ERR**

#### Niepoprawna długość komendy

- Pole Długość Komendy jest większe niż 256
- Faktyczna długość ramki nie zgadza się z deklarowaną długością pola Dane
- Ramka odrzucona: **ERR\_LENGTH**

#### Nieprawidłowe znaki w polu Dane

- Pole DANE zawiera znaki inne niż cyfry 0-9
- Ramka odrzucona: **ERR**

#### Błąd Sumy kontrolnej

- CRC-8 nie zgadza się z obliczoną wartością po dekodowaniu pola Dane.
- Ramka odrzucona: **ERR\_CRC**

#### Nieznany kod komendy

- Kod komendy (pierwsze 2 cyfry pola Dane) nie należy do zakresu 10-18
- Ramka odrzucona: **ERR: UNKNOWN CMD**

*Błędy semantyczne (odpowiedź: ramka z kodem błędu)*

*Gdy ramka jest strukturalnie poprawna, ale komenda lub jej parametry są nieprawidłowe, urządzenie wysyła ramkę odpowiedzi z odpowiednim kodem błędu w polu DATA.*

#### Brak wymaganych parametrów lub nieprawidłowe parametry

- Komenda wymaga parametrów, które nie zostały dostarczone
- Parametry mają nieprawidłową wartość
- Tryb czujnika poza zakresem 1-6
- Odpowiedź: Ramka z kodem 01 w polu Dane

#### Wartość parametru poza dopuszczalnym zakresem

- Parametr komendy SET\_INTERVAL(**14**) wskazuje interwał poza zakresem 1-9999 ms
- Odpowiedź: Ramka z kodem 02 w polu Dane

#### Brak danych do pobrania

- Komendy DOWNLOAD(12) lub VIEW(13) gdy brak pomiarów w buforze
- Offset przekracza liczbę dostępnych pomiarów
- Odpowiedź: Ramka z kodem 03 w polu Dane

### Wartość interwału spoza dopuszczalnego zakresu

- Parametr komendy SET\_INTERVAL(14) wskazuje interwał poza zakresem 1-9999 ms
- Odpowiedź: Ramka z kodem 02 w polu Dane

## 5. Implementacja w kodzie

### a. Stan maszyny parsowania ramek

```

1. typedef enum {
2.     ST_IDLE = 0,
3.     ST_COLLECT
4. } frame_state_t; // Stan maszyny parsowania ramki
5.
6. typedef struct measurement_entry_t {
7.     float lux;           // Wartość natężenia światła w luksach
8.     uint32_t timestamp; // Timestamp pomiaru
9. } measurement_entry_t;

```

### b. Definicje

```

1. #define USART_TXBUF_LEN 1512
2. #define USART_RXBUF_LEN 512
3. #define MAX_MEASUREMENTS_PER_FRAME 63

```

### c. Zmienne

```

1. Stan parsera ramek
2. frame_state_t st = ST_IDLE;
3. char frame[300];
4. uint16_t pos = 0;
5.
6. // Bufory Cykliczne USART
7. uint8_t USART_TxBuf[USART_TXBUF_LEN];
8. uint8_t USART_RxBuf[USART_RXBUF_LEN];
9.
10. // Indeksy buforów USART
11. __IO int USART_TX_Empty = 0; // Nadawanie head
12. __IO int USART_TX_Busy = 0; // Nadawanie tail
13. __IO int USART_RX_Empty = 0; // Odbieranie head
14. __IO int USART_RX_Busy = 0; // Odbieranie tail
15.
16. __IO uint8_t USART_RxBufOverflow = 0; // Flaga błędu
17. uint8_t rx_byte; // Zmienna do odbioru bajtu

```

### d. Funkcje USART

Proces rozpoczyna się w przerwaniu sprzętowym USART. System wykorzystuje **bufor cykliczny**, co pozwala na asynchroniczne odbieranie danych bez blokowania procesora.

- **Odbiór:** Za każdym razem, gdy do kontrolera trafi bajt, wywoływany jest callback HAL\_UART\_RxCpltCallback. Bajt trafia do bufora USART\_RxBuf na pozycję wskazywaną przez indeks USART\_RX\_Empty (head). Jeśli bufor jest pełny, ustawiana jest flaga błędu USART\_RxBufOverflow.
- **Pobieranie danych:** Główna pętla programu wywołuje funkcję process\_uart\_buffer(). Sprawdza ona za pomocą USART\_kbhit(), czy indeksy Empty (head) i Busy (tail) są różne, co sygnalizuje obecność nowych danych. Pobranie bajtu odbywa się przez USART\_getchar().

```

1. // Sprawdza czy są dane
2. uint8_t USART_kbhit() {
3.     if (USART_RX_Empty == USART_RX_Busy) {
4.         return 0; //Buffer is empty
5.     } else {
6.         return 1; //Buffer has data
7.     }
8. }
9.
10. // Odczytuje znak z bufora
11. int16_t USART_getchar() {
12.     int16_t tmp;
13.     if (USART_RX_Empty != USART_RX_Busy) {
14.         tmp = USART_RxBuf[USART_RX_Busy];
15.         USART_RX_Busy++;
16.         if (USART_RX_Busy >= USART_RXBUF_LEN)
17.             USART_RX_Busy = 0;
18.         return tmp;
19.     } else
20.         return -1;
21. }
22. // Wysyła sformatowany tekst
23. void USART_fsend(char *format, ...) {
24.     char tmp_rs[300]; // Zwiększone dla maksymalnej ramki (271) + margines
25.     int i;
26.     __IO int idx;
27.     va_list arglist;
28.     va_start(arglist, format);
29.     vsprintf(tmp_rs, format, arglist);
30.     va_end(arglist);
31.     idx = USART_TX_Empty;
32.     for (i = 0; i < strlen(tmp_rs); i++) {
33.         USART_TxBuf[idx] = tmp_rs[i];
34.         idx++;
35.         if (idx >= USART_TXBUF_LEN)
36.             idx = 0;
37.     }
38.     __disable_irq();
39.     if ((USART_TX_Empty == USART_TX_Busy)
40.         && (_HAL_UART_GET_FLAG(&huart2, UART_FLAG_TXE) == SET)) {
41.         USART_TX_Empty = idx;
42.         uint8_t tmp = USART_TxBuf[USART_TX_Busy];
43.         USART_TX_Busy++;
44.         if (USART_TX_Busy >= USART_TXBUF_LEN)
45.             USART_TX_Busy = 0;
46.         HAL_UART_Transmit_IT(&huart2, &tmp, 1);
47.     } else {
48.         USART_TX_Empty = idx;
49.     }
50.     __enable_irq();
51. }

```

```

1. // Callbacki
2. void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) {
3.     if (huart == &huart2) {
4.         if (USART_TX_Empty != USART_TX_Busy) {
5.             uint8_t tmp = USART_TxBuf[USART_TX_Busy];
6.             USART_TX_Busy++;
7.             if (USART_TX_Busy >= USART_TXBUF_LEN)
8.                 USART_TX_Busy = 0;
9.             HAL_UART_Transmit_IT(&huart2, &tmp, 1);
10.        }
11.    }
12. }
13. void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){

```

```

14.     if (huart == &huart2) {
15.         int next_head = (USART_RX_Empty + 1) % USART_RXBUF_LEN;
16.         if (next_head == USART_RX_Busy) {
17.             USART_RxBufOverflow = 1;
18.         } else {
19.             USART_RxBuf[USART_RX_Empty] = rx_byte;
20.             USART_RX_Empty = next_head;
21.         }
22.         HAL_UART_Receive_IT(&huart2, &rx_byte, 1);
23.     }
24. }
```

#### e. Odczyt ramki

Odebrane bajty są przetwarzane przez maszynę stanów (frame\_state\_t), która zarządza procesem składania kompletnej ramki w buforze frame[].

- **Stan ST\_IDLE:** Jest to stan spoczynkowy. Maszyna ignoruje wszystkie znaki, dopóki nie napotka znaku początku ramki '&'. Wtedy zeruje indeks pos, zapisuje znak do bufora i przechodzi w stan ST\_COLLECT.
- **Stan ST\_COLLECT:**
  - Jeśli ponownie napotka '&', uznaje, że poprzednia (niedokończona) ramka była błędna, resetuje indeks pos i zaczyna składanie nowej ramki od początku.
  - Każdy inny znak jest dopisywany do tablicy frame, o ile nie przekroczyło jej rozmiaru.
  - Napotkanie znaku końca ramki '\*' kończy proces gromadzenia. Maszyna dodaje terminator stringa (\0), wywołuje funkcję validate\_frame() i wraca do stanu ST\_IDLE

```

1. void process_uart_buffer(void)
2. {
3.     while(USART_kbhit())
4.     {
5.         char c = USART_getchar();
6.
7.         switch(st)
8.         {
9.             case ST_IDLE:
10.                 if(c == '&')
11.                 {
12.                     pos = 0;
13.                     frame[pos++] = c;
14.                     st = ST_COLLECT;
15.                 }
16.                 break;
17.
18.             case ST_COLLECT:
19.                 /* nowy start resetuje ramkę */
20.                 if(c == '&')
21.                 {
22.                     pos = 0;
23.                     frame[pos++] = c;
24.                     break;
25.                 }
26.
27.                 /* zapisuj znak */
28.                 if(pos < sizeof(frame)-1)
29.                     frame[pos++] = c;
30.
31.                 /* koniec ramki */
32.                 if(c == '*')
33.                 {
34.                     frame[pos] = 0; // string end
35.                     validate_frame(frame, pos);
36.                     st = ST_IDLE;
37.                 }
38. }
```

```

38.         break;
39.     }
40. }
41. }
```

#### f. Walidacja ramki

Funkcja validate\_frame pełni rolę "strażnika" logiki protokołu. Zanim komenda zostanie wykonana, ramka przechodzi testy:

- Długość i Format:** Sprawdzane jest, czy ramka ma minimum 13 znaków. Następnie wycinane są pola: **SRC** (źródło), **DST** (cel), **ID** (identyfikator) oraz **LEN** (deklarowana długość danych).
- Walentacja znaków:** Adresy muszą składać się z dozwolonych znaków (funkcja `is_addr_char_valid`), a pola ID i LEN wyłącznie z cyfr (`is_digits_only`).
- Zgodność Pola Danych:** System sprawdza, czy faktyczna liczba odebranych znaków odpowiada wartości zapisanej w polu LEN oraz czy dane (np. kody komend) są numeryczne.
- Suma Kontrolna CRC-8:** To kluczowy etap bezpieczeństwa. System kopiuje wszystkie pola (od SRC do DATA) do tymczasowego bufora i oblicza sumę kontrolną za pomocą wielomianu  $x^8 + x^2 + x^1 + 1$  (0x07). Obliczone `calc_crc` jest porównywane z wartością `rx_crc` otrzymaną w ramce (przekonwertowaną z formatu HEX przez `hex2byte`). Jeśli sumy się nie zgadzają, wysyłany jest komunikat `ERR_CRC` i ramka zostaje porzucona.

```

1. void validate_frame(char *f, uint16_tflen)
2. {
3.     char src[4], dst[4], id[3], len_str[4];
4.
5.     /* ===== Sprawdzenie minimalnej długości ramki ===== */
6.     if(flen < 13) {
7.         USART_fsend("ERR: TOO SHORT\r\n");
8.         return;
9.     }
10.
11.    /* ===== Parsowanie pól ===== */
12.    uint16_t pos = 1; // po '&'
13.    memcpy(src, &f[pos], 3); src[3]=0; pos+=3;
14.    memcpy(dst, &f[pos], 3); dst[3]=0; pos+=3;
15.    memcpy(id, &f[pos], 2); id[2]=0; pos+=2;
16.    memcpy(len_str, &f[pos], 3); len_str[3]=0; pos+=3;
17.
18.    /* ===== Walidacja znaków SRC/DST ===== */
19.    for (uint8_t i = 0; i < 3; i++) {
20.        if (!is_addr_char_valid(src[i]) || !is_addr_char_valid(dst[i])) {
21.            USART_fsend("ERR: INVALID ADDR\r\n");
22.            return;
23.        }
24.    }
25.
26.    /* ===== Sprawdzenie formatu ID (tylko cyfry 0-9) ===== */
27.    if(!is_digits_only(id, 2)) {
28.        USART_fsend("ERR: INVALID ID\r\n");
29.        return;
30.    }
31.
32.    /* ===== Sprawdzenie formatu długości (tylko cyfry 0-9) ===== */
33.    if(!is_digits_only(len_str, 3)) {
34.        USART_fsend("ERR\r\n");
35.        return;
36.    }
37.
38.    uint16_t data_len_declared = atoi(len_str);
39.
40.    /* ===== Sprawdzenie maksymalnej długości ===== */
```

```

41.     if(data_len_declared > 256) {
42.         USART_fsend("ERR_LENGTH\r\n");
43.         return;
44.     }
45.
46.     /* ===== Sprawdzenie faktycznej długości danych ===== */
47.     uint16_t data_start = pos;
48.     if(flen < (data_start + data_len_declared + 3)) {
49.         USART_fsend("ERR_LENGTH\r\n");
50.         return;
51.     }
52.
53.     uint16_t crc_pos = data_start + data_len_declared;
54.     uint16_t data_len_real = data_len_declared;
55.
56.     /* ===== Odczyt danych ===== */
57.     char data[257];
58.     memcpy(data, &f[data_start], data_len_real);
59.     data[data_len_real] = 0;
60.
61.     /* ===== Walidacja znaków DATA (tylko cyfry 0-9) ===== */
62.     if (!is_digits_only(data, data_len_real)) {
63.         USART_fsend("ERR\r\n");
64.         return;
65.     }
66.
67.     /* ===== Sprawdzenie czy mamy CRC ===== */
68.     if(flen < (crc_pos + 2)) {
69.         USART_fsend("ERR\r\n");
70.         return;
71.     }
72.
73.     /* ===== Odczyt i weryfikacja CRC ===== */
74.     uint8_t rx_crc = hex2byte(f[crc_pos], f[crc_pos+1]);
75.
76.     /* ===== Budowa bufora do obliczenia CRC ===== */
77.     uint8_t buf[270];
78.     uint16_t p = 0;
79.     memcpy(&buf[p], src, 3); p+=3;
80.     memcpy(&buf[p], dst, 3); p+=3;
81.     memcpy(&buf[p], id, 2); p+=2;
82.     memcpy(&buf[p], len_str, 3); p+=3;
83.     memcpy(&buf[p], data, data_len_real); p+=data_len_real;
84.
85.     /* ===== Obliczenie CRC ===== */
86.     uint8_t calc_crc = crc8(buf, p);
87.
88.     /* ===== Sprawdzenie CRC ===== */
89.     if(calc_crc != rx_crc) {
90.         USART_fsend("ERR_CRC\r\n");
91.         return;
92.     }
93.
94.     // Przekazanie informacji o ramce do obsługi komendy
95.     handle_command(data, src, dst, id);
96. }
```

#### g. Obsługa ramki

Po pomyślnej walidacji, dane trafiają do `handle_command()`. Funkcja ta interpretuje pierwsze dwie cyfry pola DATA jako kod operacji:

- **Komendy sterujące (10 - START, 11 - STOP):** Aktywują lub dezaktywują automatyczny odczyt czujnika BH1750.
- **Komendy konfiguracyjne (14 - SET\_INTERVAL, 16 - SET\_MODE):** Zmieniają parametry pracy zegara oraz czujnika (częstotliwość pracy zegara, tryb rozdzielczości I2C).

- **Komendy odczytu (12 - DOWNLOAD, 13 - VIEW):** Pobierają dane z pamięci pomiarów. W przypadku VIEW (kod 13), system potrafi spakować do 63 pomiarów w jedną ramkę, dbając o to, by nie przekroczyć limitów bufora.

**Generowanie odpowiedzi:** Każda poprawna komenda kończy się wywołaniem send\_response\_frame(). Funkcja ta buduje nową ramkę według schematu, automatycznie oblicza dla niej sumę kontrolną CRC-8, konwertuje ją na format tekstowy HEX i przekazuje do bufora nadawczego USART przez USART\_fsend.

```

1. void handle_command(char *cmd, const char *src_addr, const char *dst_addr, const char *id) {
2.     const char *device_addr = dst_addr;
3.
4.     // Zakres dozwolonych wartości interwału
5.     const uint16_t MIN_INTERVAL = 1;
6.     const uint16_t MAX_INTERVAL = 9999;
7.
8.     // Sprawdzenie minimalnej długości (kod komendy = 2 cyfry)
9.     if (strlen(cmd) < 2 || !is_digits_only(cmd, 2)) {
10.         USART_fsend("ERR: INVALID CMD\r\n");
11.         return;
12.     }
13.
14.    // Wyodrębnienie kodu komendy (pierwsze 2 cyfry)
15.    uint8_t cmd_code = (cmd[0] - '0') * 10 + (cmd[1] - '0');
16.    const char *params = (strlen(cmd) > 2) ? &cmd[2] : "";
17.
18.    // 10 - START
19.    if (cmd_code == 10) {
20.        Measurement_EnableAutoRead(1);
21.        send_response_frame(device_addr, src_addr, id, "00"); // OK
22.    }
23.    // 11 - STOP
24.    else if (cmd_code == 11) {
25.        Measurement_EnableAutoRead(0);
26.        send_response_frame(device_addr, src_addr, id, "00"); // OK
27.    }
28.    // 12 - DOWNLOAD (ostatni pomiar)
29.    else if (cmd_code == 12) {
30.        uint16_t count = Measurement_GetCount();
31.        if (count == 0) {
32.            send_response_frame(device_addr, src_addr, id, "03"); //
33.            ERR_NO_DATA
34.            return;
35.        }
36.        measurement_entry_t *entry = Measurement_GetEntry(count - 1);
37.        if (!entry) {
38.            send_response_frame(device_addr, src_addr, id, "03"); //
39.            ERR_NO_DATA
40.            return;
41.        }
42.        uint32_t lux_val = (uint32_t)(entry->lux + 0.5f);
43.        if (lux_val > 9999) {
44.            lux_val = 9999;
45.        }
46.        char response[5];
47.        response[0] = '0' + (lux_val / 1000) % 10;
48.        response[1] = '0' + (lux_val / 100) % 10;
49.        response[2] = '0' + (lux_val / 10) % 10;
50.        response[3] = '0' + lux_val % 10;
51.        response[4] = 0;
52.        send_response_frame(device_addr, src_addr, id, response);
53.    }
54.    // 13 - VIEW (+ xxzz parametry)
55.    else if (cmd_code == 13) {

```

## Obsługa czujnika BH1750 przy użyciu STM32F446RE

```

54.         if (strlen(params) < 4) {
55.             send_response_frame(device_addr, src_addr, id, "01"); // ERR_PARAM
56.             return;
57.         }
58.         uint8_t start_offset = (params[0] - '0') * 10 + (params[1] - '0');
59.         uint8_t count_req = (params[2] - '0') * 10 + (params[3] - '0');
60.         if (count_req == 0) {
61.             send_response_frame(device_addr, src_addr, id, "01"); // ERR_PARAM
62.             return;
63.         }
64.         uint16_t count = Measurement_GetCount();
65.
66.         // Sprawdzenie czy offset nie przekracza liczby pomiarów
67.         if (start_offset >= count) {
68.             send_response_frame(device_addr, src_addr, id, "03"); //
ERR_NO_DATA
69.             return;
70.         }
71.
72.         // Maksymalna liczba pomiarów na ramkę: (256 - 2) / 4 = 63
73.         // Format: xxLLLLLLLLLL... (offset 2 znaki + N*4 znaki lux)
74.         #define MAX_MEASUREMENTS_PER_FRAME 63
75.         char data_out[256];
76.         uint8_t measurements_sent = 0;
77.
78.         while (measurements_sent < count_req) {
79.             uint8_t current_offset = start_offset + measurements_sent;
80.             uint8_t batch_size = 0;
81.             uint16_t data_pos = 0;
82.
83.             // Offset (2 znaki)
84.             data_out[data_pos++] = '0' + (current_offset / 10) % 10;
85.             data_out[data_pos++] = '0' + current_offset % 10;
86.
87.             // Pakowanie pomiarów do ramki
88.             for (uint8_t i = 0; i < MAX_MEASUREMENTS_PER_FRAME &&
measurements_sent < count_req; i++) {
89.                 int32_t idx = (int32_t)count - 1 -
(int32_t)current_offset - (int32_t)i;
90.                 if (idx < 0) {
91.                     break;
92.                 }
93.                 measurement_entry_t *entry =
Measurement_GetEntry((uint16_t)idx);
94.                 if (!entry) {
95.                     break;
96.                 }
97.                 uint32_t lux_val = (uint32_t)(entry->lux + 0.5f);
98.                 if (lux_val > 9999) {
99.                     lux_val = 9999;
}
100.                // Dodanie 4-cyfrowej wartości lux
101.                data_out[data_pos++] = '0' + (lux_val / 1000) % 10;
102.                data_out[data_pos++] = '0' + (lux_val / 100) % 10;
103.                data_out[data_pos++] = '0' + (lux_val / 10) % 10;
104.                data_out[data_pos++] = '0' + lux_val % 10;
105.                batch_size++;
106.                measurements_sent++;
107.            }
108.        }
109.
110.        data_out[data_pos] = 0;
111.        send_response_frame(device_addr, src_addr, id, data_out);
112.
113.        // Jeśli nie wysłano żadnych pomiarów, przerwij
114.        if (batch_size == 0) {
115.            break;
116.        }
117.    }
118. }

```

## Obsługa czujnika BH1750 przy użyciu STM32F446RE

```

119. // 14 - SET_INTERVAL (+ xxxx parametr)
120. else if (cmd_code == 14) {
121.     if (strlen(params) < 4) {
122.         send_response_frame(device_addr, src_addr, id, "01"); // ERR_PARAM
123.         return;
124.     }
125.     uint16_t interval_ms = atoi(params);
126.     if (interval_ms < MIN_INTERVAL || interval_ms > MAX_INTERVAL) {
127.         send_response_frame(device_addr, src_addr, id, "02"); // ERR_RANGE
128.         return;
129.     }
130.     Measurement_SetInterval(interval_ms);
131.     send_response_frame(device_addr, src_addr, id, "00"); // OK
132. }
133. // 15 - GET_INTERVAL
134. else if (cmd_code == 15) {
135.     uint16_t interval_ms = measurement_auto.interval_ms;
136.     if (interval_ms > 9999) {
137.         interval_ms = 9999;
138.     }
139.     char response[6];
140.     response[0] = '0' + (interval_ms / 1000) % 10;
141.     response[1] = '0' + (interval_ms / 100) % 10;
142.     response[2] = '0' + (interval_ms / 10) % 10;
143.     response[3] = '0' + interval_ms % 10;
144.     response[4] = 0;
145.     send_response_frame(device_addr, src_addr, id, response);
146. }
147. // 16 - SET_MODE (+ x parametr)
148. else if (cmd_code == 16) {
149.     if (strlen(params) < 1 || params[0] < '1' || params[0] > '6') {
150.         send_response_frame(device_addr, src_addr, id, "01"); // ERR_PARAM
151.         return;
152.     }
153.     uint8_t mode_num = params[0] - '0';
154.     uint8_t mode_value;
155.     switch (mode_num) {
156.         case 1: mode_value = BH1750_CONTINUOUS_HIGH_RES_MODE; break;
157.         case 2: mode_value = BH1750_CONTINUOUS_HIGH_RES_MODE_2; break;
158.         case 3: mode_value = BH1750_CONTINUOUS_LOW_RES_MODE; break;
159.         case 4: mode_value = BH1750_ONETIME_HIGH_RES_MODE; break;
160.         case 5: mode_value = BH1750_ONETIME_HIGH_RES_MODE_2; break;
161.         case 6: mode_value = BH1750_ONETIME_LOW_RES_MODE; break;
162.         default:
163.             send_response_frame(device_addr, src_addr, id, "01"); // ERR_PARAM
164.             return;
165.     }
166.     HAL_StatusTypeDef status = BH1750_SetMode(mode_value);
167.     if (status == HAL_OK) {
168.         send_response_frame(device_addr, src_addr, id, "00"); // OK
169.     } else {
170.         send_response_frame(device_addr, src_addr, id, "04"); // ERR_I2C
171.     }
172. }
173. // 17 - GET_MODE
174. else if (cmd_code == 17) {
175.     char mode_char = '1';
176.     switch (bh1750_current_mode) {
177.         case BH1750_CONTINUOUS_HIGH_RES_MODE: mode_char = '1'; break;
178.         case BH1750_CONTINUOUS_HIGH_RES_MODE_2: mode_char = '2'; break;
179.         case BH1750_CONTINUOUS_LOW_RES_MODE: mode_char = '3'; break;
180.         case BH1750_ONETIME_HIGH_RES_MODE: mode_char = '4'; break;
181.         case BH1750_ONETIME_HIGH_RES_MODE_2: mode_char = '5'; break;
182.         case BH1750_ONETIME_LOW_RES_MODE: mode_char = '6'; break;
183.         default: mode_char = '1'; break;
184.     }
185.     char response[2];
186.     response[0] = mode_char;

```

```

187.         response[1] = 0;
188.         send_response_frame(device_addr, src_addr, id, response);
189.     }
190. // Nieznany kod komendy
191. else {
192.     USART_fsend("ERR: UNKNOWN CMD\r\n");
193. }
194. }
```

#### h. Funkcje pomocnicze

```

1. // Konwersja znaków hex na bajt
2. uint8_t hex2byte(char hi, char lo) {
3.     uint8_t high = (hi >= '0' && hi <= '9') ? hi - '0' :
4.             (hi >= 'A' && hi <= 'F') ? hi - 'A' + 10 :
5.                 (hi >= 'a' && hi <= 'f') ? hi - 'a' + 10 : 0;
6.     uint8_t low = (lo >= '0' && lo <= '9') ? lo - '0' :
7.             (lo >= 'A' && lo <= 'F') ? lo - 'A' + 10 :
8.                 (lo >= 'a' && lo <= 'f') ? lo - 'a' + 10 : 0;
9.     return (high << 4) | low;
10. }

1. // Obliczanie sumy kontrolnej CRC-8
uint8_t crc8(uint8_t *data, uint16_t len) {
2.     uint8_t crc = 0x00;
3.     for (uint16_t i = 0; i < len; i++) {
4.         crc ^= data[i];
5.         for (uint8_t j = 0; j < 8; j++) {
6.             if (crc & 0x80)
7.                 crc = (crc << 1) ^ 0x07;
8.             else
9.                 crc <<= 1;
10.        }1.
11.    }
12.    return crc;
13. }

1. // Konwersja bajt na znaki hex
void byte2hex(uint8_t byte, char *hex) {
2.     static const char hex_chars[] = "0123456789ABCDEF";
3.     hex[0] = hex_chars[(byte >> 4) & 0x0F];
4.     hex[1] = hex_chars[byte & 0x0F];
5. }

1. // Walidacja czy string zawiera tylko cyfry
uint8_t is_digits_only(const char *str, uint16_t len) {
2.     for (uint16_t i = 0; i < len; i++) {
3.         if (str[i] < '0' || str[i] > '9') {
4.             return 0;
5.         }
6.     }
7.     return 1;
8. }

1. // Walidacja znaku adresu
static uint8_t is_addr_char_valid(char c) {
2.     return (c >= 0x21 && c <= 0x7E && c != '&' && c != '*');
3. }
```

```

1. // Budowanie i wysyłanie ramki odpowiedzi
2. void send_response_frame(const char *src_addr, const char *dst_addr, const char *id, const
char *data) {
3.     char frame[271];
4.     uint16_t pos = 0;
5.     uint8_t crc_buf[300];
6.     uint16_t crc_pos = 0;
7.
8.     // Budowanie ramki: & SRC DST ID LEN DATA CRC *
9.     frame[pos++] = '&';
10.
11.    // SRC (3 znaki)
```

```

12.     memcpy(&frame[pos], src_addr, 3);
13.     pos += 3;
14.     memcpy(&crc_buf[crc_pos], src_addr, 3);
15.     crc_pos += 3;
16.
17.     // DST (3 znaki)
18.     memcpy(&frame[pos], dst_addr, 3);
19.     pos += 3;
20.     memcpy(&crc_buf[crc_pos], dst_addr, 3);
21.     crc_pos += 3;
22.
23.     // ID (2 znaki)
24.     memcpy(&frame[pos], id, 2);
25.     pos += 2;
26.     memcpy(&crc_buf[crc_pos], id, 2);
27.     crc_pos += 2;
28.
29.     // LEN (3 znaki) - długość danych
30.     uint16_t data_len = strlen(data);
31.     if (data_len > 256) {
32.         return;
33.     }
34.     char len_str[4];
35.     len_str[0] = '0' + (data_len / 100) % 10;
36.     len_str[1] = '0' + (data_len / 10) % 10;
37.     len_str[2] = '0' + data_len % 10;
38.     len_str[3] = 0;
39.     memcpy(&frame[pos], len_str, 3);
40.     pos += 3;
41.     memcpy(&crc_buf[crc_pos], len_str, 3);
42.     crc_pos += 3;
43.
44.     // DATA
45.     memcpy(&frame[pos], data, data_len);
46.     pos += data_len;
47.     memcpy(&crc_buf[crc_pos], data, data_len);
48.     crc_pos += data_len;
49.
50.     // Obliczanie CRC
51.     uint8_t crc = crc8(crc_buf, crc_pos);
52.     char crc_hex[3];
53.     byte2hex(crc, crc_hex);
54.     crc_hex[2] = 0;
55.
56.     // CRC (2 znaki hex)
57.     memcpy(&frame[pos], crc_hex, 2);
58.     pos += 2;
59.
60.     // Zakończenie ramki
61.     frame[pos++] = '*';
62.     frame[pos] = 0;
63.
64.     // Wysyłanie ramki przez USART
65.     USART_fsend("%s", frame);
66. }
```

### 3. Parametry Komunikacji

*Konfiguracja USART2*

- Baud Rate – 115200 bits/s
- Word Length – 8 bitów
- Parity – brak
- Stop Bits – 1

- Data Direction – Receive and Transmit
- OverSampling – 16

#### *Konfiguracja I2C1*

- I2C Speed Mode – Standard Mode
- I2C Clock Speed – 100000
- Addressing Mode – 7-bit
- Dual Address Mode – disabled
- General Call Mode – disabled
- Clock No Stretch Mode – disabled

#### *Konfiguracja TIM2*

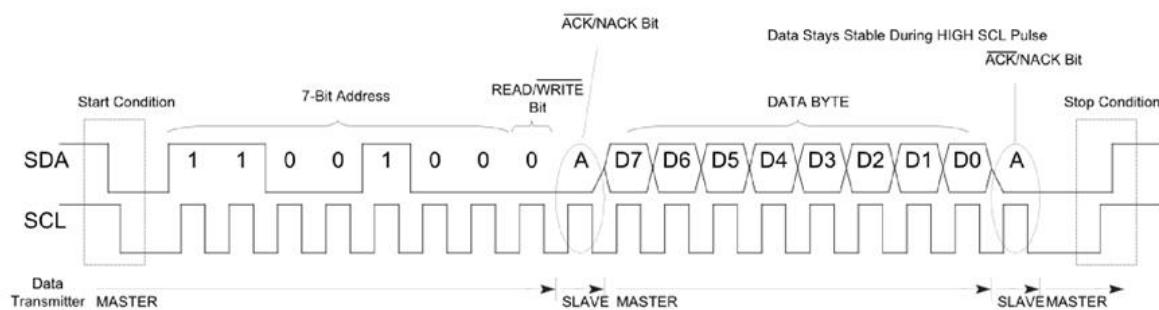
- Clock Source – Internal Clock
- Prescaler – 8999
- Counter Mode – Up
- Counter Period (ARR) – 9
- Clock Division – No Division
- Auto-Reload Preload – Enable
- Częstotliwość przerwań – 1 kHz (1 ms)

## 4. Opis działania czujnika BH1750

### 1. Interfejs komunikacyjny czujnika BH1750

<b>VCC</b>	<b>3.3V</b>
<b>GND</b>	<b>GND</b>
<b>SCL</b>	<b>PB6</b>
<b>SDA</b>	<b>PB7</b>
<b>ADDR</b>	<b>0x23 lub 0x5C</b>

Adres czujnika zależy od stanu, dla stanu niskiego jest 0x23, a dla wysokiego 0x5C. Domyslnie jest w stanie niskim.



## 2. Konfiguracja początkowa systemu

### 1. Struktura danych i definicje

```

1. // Adresy I2C czujnika BH1750 (7-bit)
2. #define BH1750_ADDR_LOW    0x23 // ADDR pin do GND
3. #define BH1750_ADDR_HIGH   0x5C // ADDR pin do VCC
4.
5. // Komendy czujnika BH1750
6. #define BH1750_CONTINUOUS_HIGH_RES_MODE     0x10 // Ciągły wysokiej rozdzielczości (1lx, 120ms)
7. #define BH1750_CONTINUOUS_HIGH_RES_MODE_2   0x11 // Ciągły wysokiej rozdzielczości 2 (0.5lx, 120ms)
8. #define BH1750_CONTINUOUS_LOW_RES_MODE      0x13 // Ciągły niskiej rozdzielczości (4lx, 16ms)
9.
10. // Zmienne globalne
11. static uint8_t bh1750_current_mode = BH1750_CONTINUOUS_HIGH_RES_MODE;
12. static uint8_t bh1750_initialized = 0;
13. static uint8_t bh1750_addr = BH1750_ADDR_LOW;
14.
15. // Stany maszyny stanów inicjalizacji
16. typedef enum {
17.     BH1750_INIT_IDLE = 0,
18.     BH1750_INIT_PWRON,
19.     BH1750_INIT_RESET,
20.     BH1750_INIT_MODE,
21.     BH1750_INIT_DONE
22. } bh1750_init_state_t;
23.
24. static bh1750_init_state_t bh1750_init_state = BH1750_INIT_IDLE;

```

System przechowuje aktualny tryb pracy czujnika w zmiennej `bh1750\_current\_mode`, flagę inicjalizacji w `bh1750\_initialized` oraz adres I2C w `bh1750\_addr`. Maszyna stanów używa typu wyliczeniowego `bh1750\_init\_state\_t` do śledzenia postępu inicjalizacji.

### 2. Struktura operacji I2C

Nieblokująca komunikacja I2C wykorzystuje strukturę `i2c\_operation\_t`, która śledzi parametry bieżącej operacji:

```

1. // Struktura dla operacji I2C
2. typedef struct {
3.     uint8_t address;      // Adres urządzenia I2C (7-bit)
4.     uint8_t *data;        // Wskaźnik na bufor danych
5.     uint16_t len;         // Długość danych
6.     uint8_t operation;   // 0 = TX (transmisja), 1 = RX (odbiór)
7.     uint8_t pending;      // Flaga: 1 = operacja w toku, 0 = wolne
8. } i2c_operation_t;
9.
10. static i2c_operation_t i2c_op = {0};

```

Flaga `pending` działa jak mutex - zapobiega rozpoczęciu nowej operacji I2C gdy poprzednia jest jeszcze aktywna.

### 3. Timer systemowy dla nieblokujących opóźnień

System wykorzystuje timer TIM2 do generowania znaczników czasowych co 1ms:

```

1. // Timer aplikacyjny oparty o TIM2 (1ms)
2. static __IO uint32_t app_tick = 0;
3.
4. static uint32_t App_GetTick(void) {
5.     return app_tick;
6. }
7.
8. // Callback przerwania timera (wywoływany co 1ms)
9. void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {

```

```

10.     if (htim->Instance == TIM2) {
11.         app_tick++;
12.     }
13. }
```

Funkcja `App\_GetTick()` zwraca liczbę milisekund od startu systemu, pozwalając na nieblokujące odmierzanie czasu.

#### 4. Struktura śledzenia czasu oczekiwania BH1750

Czujnik BH1750 wymaga określonego czasu na wykonanie pomiaru (120ms lub 16ms w zależności od trybu). System śledzi ten czas bez blokowania procesora:

```

1. // Struktura dla śledzenia czasu oczekiwania BH1750
2. typedef struct {
3.     uint32_t start_time; // Czas rozpoczęcia (App_GetTick())
4.     uint32_t wait_time; // Wymagany czas oczekiwania w ms
5.     uint8_t active; // 1 = aktywne oczekiwanie, 0 = zakończone
6. } bh1750_timing_t;
7.
8. static bh1750_timing_t bh1750_timing = {0};
9.
10. void BH1750_StartTiming(uint32_t wait_time_ms) {
11.     bh1750_timing.start_time = App_GetTick();
12.     bh1750_timing.wait_time = wait_time_ms;
13.     bh1750_timing.active = 1;
14. }
15.
16. uint8_t BH1750_IsTimingReady(void) {
17.     if (!bh1750_timing.active) {
18.         return 1; // Nie ma aktywnego oczekiwania
19.     }
20.
21.     uint32_t current_time = App_GetTick();
22.     if ((current_time - bh1750_timing.start_time) >= bh1750_timing.wait_time) {
23.         bh1750_timing.active = 0;
24.         return 1; // Czas minął
25.     }
26.
27.     return 0; // Jeszcze czeka
28. }
```

### 3. Inicjalizacja czujnika przez maszyne stanów

#### 1. Nieblokująca transmisja I2C

Funkcja `I2C\_Transmit\_IT()` inicjuje transmisję przez przerwania:

```

1. HAL_StatusTypeDef I2C_Transmit_IT(uint8_t address, uint8_t *data, uint16_t len) {
2.     if (i2c_op.pending) {
3.         return HAL_BUSY; // Magistrala zajęta
4.     }
5.
6.     if (len > I2C_TXBUF_LEN) {
7.         return HAL_ERROR; // Zbyt duże dane
8.     }
9.
10.    // Sprawdź stan HAL I2C
11.    HAL_I2C_StateTypeDef state = HAL_I2C_GetState(&i2c1);
12.    if (state != HAL_I2C_STATE_READY) {
13.        HAL_I2C_DeInit(&i2c1);
14.        HAL_I2C_Init(&i2c1);
15.        return HAL_ERROR;
16.    }
17.
18.    // Kopiowanie danych do bufora
19.    for (uint16_t i = 0; i < len; i++) {
20.        I2C_TxBuf[i] = data[i];
21.    }
22.
23.    i2c_op.address = address;
24.    i2c_op.data = I2C_TxBuf;
```

```

25.     i2c_op.len = len;
26.     i2c_op.operation = 0; // TX
27.     i2c_op.pending = 1; // Zaznacz zajętość
28.
29.     // Rozpoczęcie transmisji przez przerwania (adres przesunięty o 1 bit w lewo)
30.     HAL_StatusTypeDef status = HAL_I2C_Master_Transmit_IT(&hi2c1, address << 1, I2C_TxBuf,
len);
31.
32.     if (status != HAL_OK) {
33.         i2c_op.pending = 0; // Zwolnij w przypadku błędu
34.     }
35.
36.     return status;
37. }
```

Funkcja ustawia flagę `i2c\_op.pending = 1`, uruchamia transmisję i natychmiast zwraca. Rzeczywista komunikacja odbywa się w tle.

### 2. Callback zakończenia transmisji I2C

Gdy transmisja się zakończy, biblioteka HAL wywołuje przerwanie:

```

1. void HAL_I2C_MasterTxCpltCallback(I2C_HandleTypeDef *hi2c) {
2.     if (hi2c == &hi2c1) {
3.         i2c_op.pending = 0; // Zwolnij magistralę
4.         // Operacja zakończona - można wykonać kolejną
5.     }
6. }
```

Callback wykonuje się w kontekście przerwania (kilka mikrosekund) i tylko zeruje flagę, sygnalizując że magistrala jest wolna.

### 3. Maszyna stanów inicjalizacji BH1750

Funkcja `BH1750\_Init\_Process()` jest wywoływana cyklicznie w pętli głównej i realizuje inicjalizację krok po kroku:

```

1. void BH1750_Init_Process(void) {
2.     static uint32_t last_retry_time = 0;
3.
4.     if (bh1750_initialized) {
5.         return; // Już zainicjalizowany
6.     }
7.
8.     // Inicjuj tylko gdy automatyczny pomiar jest włączony
9.     if (!measurement_auto.enabled) {
10.        return;
11.    }
12.
13.    // Jeśli był błąd I2C, zresetuj stan inicjalizacji
14.    if (I2C_Error) {
15.        I2C_Error = 0;
16.        bh1750_initialized = 0;
17.        bh1750_init_state = BH1750_INIT_IDLE;
18.        return;
19.    }
20.
21.    // Nie rozpoczynaj kolejnego kroku, jeśli I2C jest zajęte
22.    if (i2c_op.pending) {
23.        return;
24.    }
25.
26.    // Throttling - nie próbuj zbyt często (1 sekunda między próbami)
27.    if (bh1750_init_state == BH1750_INIT_IDLE) {
28.        uint32_t now = App_GetTick();
29.        if (now - last_retry_time < 1000) {
30.            return; // Czekaj
31.        }
32.        last_retry_time = now;
33.    }
34.
35.    switch (bh1750_init_state) {
```

```

36.     case BH1750_INIT_IDLE: {
37.         // Krok 1: Wysłanie komendy POWER_ON (0x01)
38.         uint8_t cmd = 0x01;
39.         HAL_StatusTypeDef status = I2C_Transmit_IT(bh1750_addr, &cmd, 1);
40.         if (status == HAL_OK) {
41.             bh1750_init_state = BH1750_INIT_PWRON;
42.         }
43.         break;
44.     }
45.     case BH1750_INIT_PWRON: {
46.         // Krok 2: Wysłanie komendy RESET (0x07)
47.         uint8_t cmd = 0x07;
48.         if (I2C_Transmit_IT(bh1750_addr, &cmd, 1) == HAL_OK) {
49.             bh1750_init_state = BH1750_INIT_RESET;
50.         }
51.         break;
52.     }
53.     case BH1750_INIT_RESET: {
54.         // Krok 3: Ustawienie trybu pomiaru
55.         if (BH1750_SetMode(bh1750_current_mode) == HAL_OK) {
56.             bh1750_init_state = BH1750_INIT_MODE;
57.         }
58.         break;
59.     }
60.     case BH1750_INIT_MODE:
61.         // Krok 4: Czekanie na zakończenie transmisji trybu
62.         if (!i2c_op.pending) {
63.             bh1750_initialized = 1;
64.             bh1750_init_state = BH1750_INIT_DONE;
65.         }
66.         break;
67.     default:
68.         break;
69.     }
70. }

```

Każdy przypadek `switch` wykonuje jeden krok i kończy działanie. W kolejnym przebiegu pętli funkcja kontynuuje od następnego stanu.

#### 4. Ustawienie trybu pomiaru

Funkcja `BH1750\_SetMode()` wysyła komendę trybu i uruchamia licznik czasu oczekiwania:

```

1. HAL_StatusTypeDef BH1750_SetMode(uint8_t mode) {
2.     HAL_StatusTypeDef status;
3.     uint8_t addr = bh1750_addr;
4.
5.     // Wysłanie komendy trybu do czujnika przez przerwania
6.     status = I2C_Transmit_IT(addr, &mode, 1);
7.
8.     if (status == HAL_OK) {
9.         bh1750_current_mode = mode;
10.
11.        // Rozpoczęcie śledzenia czasu oczekiwania
12.        // Opóźnienie w zależności od trybu (120ms dla high res, 16ms dla low res)
13.        if (mode == BH1750_CONTINUOUS_LOW_RES_MODE || mode == BH1750_ONETIME_LOW_RES_MODE) {
14.            BH1750_StartTiming(16);
15.        } else {
16.            BH1750_StartTiming(120);
17.        }
18.    }
19.
20.    return status;
21. }

```

Funkcja uruchamia nieblokujące odmierzanie 120ms (lub 16ms), aby system wiedział kiedy czujnik będzie gotowy do pierwszego odczytu.

### 5. Obsługa błędów I2C

Gdy wystąpi błąd komunikacji, biblioteka HAL wywołuje callback:

```
1. void HAL_I2C_ErrorCallback(I2C_HandleTypeDef *hi2c) {
2.     if (hi2c == &hi2c1) {
3.         I2C_Error = 1;
4.         i2c_op.pending = 0;
5.         I2C_BusReset_Pending = 1; // Uruchom procedurę recovery magistrali
6.     }
7. }
```

Flaga `I2C\_Error` powoduje reset maszyny stanów w `BH1750\_Init\_Process()`, co pozwala na automatyczną ponowną próbę inicjalizacji.

## 4. Automatyczne odczyty pomiarów

### 1. Struktura automatycznego odczytu

System wykorzystuje strukturę `measurement\_auto\_t` do zarządzania automatycznymi pomiarami:

```
1. // Struktura dla automatycznego odczytu pomiarów
2. typedef struct {
3.     uint32_t interval_ms;          // Interwał pomiarowy w ms
4.     uint32_t last_measurement;    // Czas ostatniego pomiaru (App_GetTick())
5.     uint8_t enabled;              // Czy automatyczny odczyt jest włączony
6. } measurement_auto_t;
7.
8. static measurement_auto_t measurement_auto = {
9.     .interval_ms = 1000,           // Domyślnie 1 sekunda
10.    .last_measurement = 0,
11.    .enabled = 0                 // Wyłączone domyślnie
12. };
13.
14. void Measurement_EnableAutoRead(uint8_t enable) {
15.     measurement_auto.enabled = enable;
16.     if (enable) {
17.         measurement_auto.last_measurement = App_GetTick();
18.     }
19. }
```

Gdy system odbierze komendę START (10), wywołuje `Measurement\_EnableAutoRead(1)`, uruchamiając automatyczne pomiary.

### 2. Proces automatycznego odczytu

Funkcja `Measurement\_AutoRead\_Process()` jest wywoływana w każdym przebiegu pętli głównej:

```
1. void Measurement_AutoRead_Process(void) {
2.     if (!measurement_auto.enabled) {
3.         return; // Automatyczny odczyt wyłączony
4.     }
5.     if (!bh1750_initialized) {
6.         return; // Czekaj na zakończenie inicjalizacji BH1750
7.     }
8.
9.     // Sprawdzenie czy dane z odczytu są gotowe i zapisanie ich
10.    if (bh1750_read_ready) {
11.        // Zapis pomiaru do bufora
12.        Measurement_AddEntry(bh1750_last_lux);
13.        bh1750_read_ready = 0; // Wyzeruj flagę po zapisaniu
14.    }
15.
16.    // Sprawdzenie czy minął interwał
17.    uint32_t current_time = App_GetTick();
18.    if ((current_time - measurement_auto.last_measurement) >= measurement_auto.interval_ms) {
19.        measurement_auto.last_measurement = current_time; // Zaktualizuj czas
20.
21.        // Sprawdzenie czy I2C nie jest zajęty
22.        if (!i2c_op.pending) {
23.            // Rozpoczęcie nowego odczytu z czujnika
24.            bh1750_read_ready = 0; // Wyzeruj flagę przed rozpoczęciem odczytu
25.        }
26.    }
27. }
```

## Obsługa czujnika BH1750 przy użyciu STM32F446RE

```

25.         HAL_StatusTypeDef status = BH1750_ReadLight(&bh1750_last_lux);
26.
27.         // Jeśli I2C zawiódł, dodaj wartość 0 jako wskaźnik błędu
28.         if (status != HAL_OK) {
29.             Measurement_AddEntry(0.0f); // Błąd I2C
30.         }
31.     }
32. }
33. }
```

Funkcja najpierw sprawdza czy poprzedni odczyt się zakończył (flaga `bh1750\_read\_ready`) i jeśli tak, zapisuje wynik do bufora. Następnie sprawdza czy minął interwał i jeśli magistrala I2C jest wolna, uruchamia nowy odczyt.

### 3. Nieblokujący odczyt z czujnika

Funkcja `BH1750\_ReadLight()` inicjuje odczyt przez przerwania:

```

1. // Bufory dla odczytu BH1750
2. static uint8_t bh1750_read_buffer[2] = {0};
3. static float bh1750_last_lux = 0.0f;
4. static uint8_t bh1750_read_ready = 0;
5.
6. HAL_StatusTypeDef BH1750_ReadLight(float *lux) {
7.     uint8_t addr = bh1750_addr;
8.
9.     // Sprawdzenie czy operacja I2C nie jest w toku
10.    if (i2c_op.pending) {
11.        return HAL_BUSY;
12.    }
13.
14.    // Sprawdzenie czy ostatni odczyt jest gotowy
15.    if (bh1750_read_ready) {
16.        *lux = bh1750_last_lux;
17.        bh1750_read_ready = 0;
18.        return HAL_OK;
19.    }
20.
21.    // Rozpoczęcie odczytu przez przerwania
22.    HAL_StatusTypeDef status = I2C_Receive_IT(addr, bh1750_read_buffer, 2);
23.
24.    return status; // Wartość zostanie przetworzona w callbacku
25. }
26.
```

### 4. Nieblokujący odbiór I2C

Funkcja `I2C\_Receive\_IT()` uruchamia odbiór danych:

```

1. HAL_StatusTypeDef I2C_Receive_IT(uint8_t address, uint8_t *data, uint16_t len) {
2.     if (i2c_op.pending) {
3.         return HAL_BUSY; // Operacja w toku
4.     }
5.
6.     if (len > I2C_RXBUF_LEN) {
7.         return HAL_ERROR; // Zbyt duże dane
8.     }
9.
10.    i2c_op.address = address;
11.    i2c_op.data = data;           // Wskaźnik na bh1750_read_buffer
12.    i2c_op.len = len;            // 2 bajty
13.    i2c_op.operation = 1;        // RX (odbiór)
14.    i2c_op.pending = 1;          // Zajmij magistralę
15.
16.    // Rozpoczęcie odbioru przez przerwania
17.    // (address << 1) | 0x01 - adres z bitem odczytu
18.    HAL_StatusTypeDef status = HAL_I2C_Master_Receive_IT(&i2c1, (address << 1) | 0x01,
19. I2C_RxBuf, len);
```

## Obsługa czujnika BH1750 przy użyciu STM32F446RE

```

20.     if (status != HAL_OK) {
21.         i2c_op.pending = 0; // Zwolnij w przypadku błędu
22.     }
23.
24.     return status;
25. }
```

Mikrokontroler wysyła na magistralę I2C adres czujnika (0x23 przesunięty o 1 bit = 0x46) z ustawionym bitem odczytu (0x01), co daje 0x47. Czujnik odpowiada ACK i zaczyna przesyłać 2 bajty danych.

### 5. Callback odbioru danych I2C

Gdy I2C odbierze oba bajty, wywołuje się callback:

```

1. void HAL_I2C_MasterRxCpltCallback(I2C_HandleTypeDef *hi2c) {
2.     if (hi2c == &hi2c1) {
3.         // Kopiowanie danych z bufora do miejsca docelowego
4.         if (i2c_op.operation == 1 && i2c_op.data != NULL) {
5.             for (uint16_t i = 0; i < i2c_op.len; i++) {
6.                 i2c_op.data[i] = I2C_RxBuf[i];
7.             }
8.
9.             // Jeśli to odczyt BH1750 (2 bajty)
10.            if (i2c_op.len == 2 && i2c_op.data == bh1750_read_buffer) {
11.                // Konwersja bajtów na 16-bitową wartość (big-endian)
12.                uint16_t raw_value = (bh1750_read_buffer[0] << 8) | bh1750_read_buffer[1];
13.
14.                // Przeliczenie na luksy: wartość / 1.2 (dla trybu H_RES_MODE)
15.                bh1750_last_lux = raw_value / 1.2f;
16.
17.                // Ustaw flagę gotowości
18.                bh1750_read_ready = 1;
19.            }
20.        }
21.        i2c_op.pending = 0; // Zwolnij magistralę
22.    }
23. }
```

Callback wykrywa że to odczyt z BH1750 (2 bajty), łączy oba bajty w wartość 16-bitową, dzieli przez 1.2 i ustawia flagęgotowości. Cała operacja trwa kilka mikrosekund.

### 6. Struktura automatycznego odczytu

W następnym przebiegu pętli głównej, `Measurement\_AutoRead\_Process()` wykrywa `bh1750\_read\_ready = 1` i zapisuje pomiar:

```

1. #define MEASUREMENT_BUFFER_SIZE 1000
2. static measurement_entry_t measurement_buffer[MEASUREMENT_BUFFER_SIZE];
3. static uint16_t measurement_write_index = 0; // Indeks do zapisu (head)
4. static uint16_t measurement_count = 0; // Liczba zapisanych pomiarów
5.
6. typedef struct measurement_entry_t {
7.     float lux; // Wartość natężenia światła w luksach
8.     uint32_t timestamp; // Timestamp pomiaru (App_GetTick())
9. } measurement_entry_t;
10.
11. void Measurement_AddEntry(float lux) {
12.     // Zapisanie pomiaru do bufora
13.     measurement_buffer[measurement_write_index].lux = lux;
14.     measurement_buffer[measurement_write_index].timestamp = App_GetTick();
15.
16.     // Aktualizacja indeksu (bufor cykliczny)
17.     measurement_write_index++;
18.     if (measurement_write_index >= MEASUREMENT_BUFFER_SIZE) {
19.         measurement_write_index = 0; // Zawinięcie bufora
20.     }
21.
22.     // Aktualizacja licznika (max 1000)
```

```
23.     if (measurement_count < MEASUREMENT_BUFFER_SIZE) {  
24.         measurement_count++;  
25.     }  
26. }
```

Bufor cykliczny może przechować do 1000 pomiarów. Po zapełnieniu najstarsze pomiary są nadpisywane.