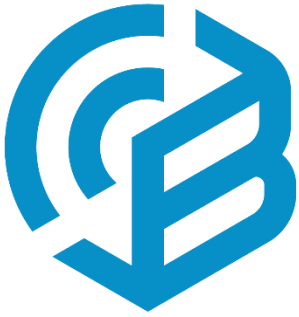


POLITECHNIKA BYDGOSKA
Wydział Telekomunikacji, Informatyki i Elektrotechniki

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

1. Oprogramować komunikacje 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 interwal czasy zdefiniowany w milisekundach uint32 i zapisywać do bufora cyklicznego o pojemności min 1000 wpisów umożliwić odczyt danych bieżących i archiwalnych.

2. Protokół komunikacyjny

1. Ramka

2. Kontrola poprawności

Każda ramka kończ y 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 (offset, który najnowszy z kolei)
- 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: xxLLLLLLLL...
- 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

```
typedef enum {
    ST_IDLE = 0,
    ST_COLLECT
} frame_state_t;  // Stan maszyny parsowania ramki

typedef struct measurement_entry_t {
      float lux;           // Wartość natężenia światła w luksach
      uint32_t timestamp;  // Timestamp pomiaru
} measurement_entry_t;
```

b. Definicje

```
#define USART_TXBUF_LEN 1512
#define USART_RXBUF_LEN 512
#define MAX_MEASUREMENTS_PER_FRAME 63
```

c. Zmienne

```
Stan parsera ramek
frame_state_t st = ST_IDLE;
char frame[300];
uint16_t pos = 0;

// Bufory Cykliczne USART
uint8_t USART_TxBuf[USART_TXBUF_LEN];
uint8_t USART_RxBuf[USART_RXBUF_LEN];

// Indeksy buforów USART
__IO int USART_TX_Empty = 0; // Nadawanie head
__IO int USART_TX_Busy = 0;  // Nadawanie tail
__IO int USART_RX_Empty = 0; // Odbieranie head
__IO int USART_RX_Busy = 0;  // Odbieranie tail

__IO uint8_t USART_RxBufOverflow = 0; // Flaga błędu
uint8_t rx_byte; // Zmienna do odbiory bajtu
```

d. Funckje USART

Proces rozpoczyna się w przerwaniu sprzętowym UART. 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().

```
// Sprawdza czy są dane
uint8_t USART_kbhit() {
    if (USART_RX_Empty == USART_RX_Busy) {
        return 0; //Buffer is empty
    } else {
        return 1; //Buffer has data
    }
}

// Odczytuje znak z bufora
int16_t USART_getchar() {
    int16_t tmp;
    if (USART_RX_Empty != USART_RX_Busy) {
        tmp = USART_RxBuf[USART_RX_Busy];
        USART_RX_Busy++;
        if (USART_RX_Busy >= USART_RXBUF_LEN)
            USART_RX_Busy = 0;
        return tmp;
    } else
        return -1;
}
// Wysyła sformatowany tekst
void USART_fsend(char *format, ...) {
    char tmp_rs[300];  // Zwiększone dla maksymalnej ramki (271) + margines
    int i;
    __IO int idx;
    va_list arglist;
    va_start(arglist, format);
    vsprintf(tmp_rs, format, arglist);
    va_end(arglist);
    idx = USART_TX_Empty;
    for (i = 0; i < strlen(tmp_rs); i++) {
        USART_TxBuf[idx] = tmp_rs[i];
        idx++;
        if (idx >= USART_TXBUF_LEN)
            idx = 0;
    }
    __disable_irq();
    if ((USART_TX_Empty == USART_TX_Busy)
            && (__HAL_UART_GET_FLAG(&huart2,UART_FLAG_TXE) == SET)) {
        USART_TX_Empty = idx;
        uint8_t tmp = USART_TxBuf[USART_TX_Busy];
        USART_TX_Busy++;
        if (USART_TX_Busy >= USART_TXBUF_LEN)
            USART_TX_Busy = 0;
        HAL_UART_Transmit_IT(&huart2, &tmp, 1);
    } else {
        USART_TX_Empty = idx;
    }
    __enable_irq();
}
```

```
// Callbacki
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) {
    if (huart == &huart2) {
        if (USART_TX_Empty != USART_TX_Busy) {
            uint8_t tmp = USART_TxBuf[USART_TX_Busy];
            USART_TX_Busy++;
            if (USART_TX_Busy >= USART_TXBUF_LEN)
                USART_TX_Busy = 0;
            HAL_UART_Transmit_IT(&huart2, &tmp, 1);
        }
    }
}
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
```

```
    if (huart == &huart2) {
        int next_head = (USART_RX_Empty + 1) % USART_RXBUF_LEN;
        if (next_head == USART_RX_Busy) {
            USART_RxBufOverflow = 1;
        } else {
            USART_RxBuf[USART_RX_Empty] = rx_byte;
            USART_RX_Empty = next_head;
        }
        HAL_UART_Receive_IT(&huart2, &rx_byte, 1);
    }
}
```

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 przekroczono jej rozmiaru.
  - Napotkanie znaku końca ramki **'*'** kończy proces gromadzenia. Maszyna dodaje terminator stringa (\0), wywołuje funkcję validate_frame() i wraca do stanu ST_IDLE

```
void process_uart_buffer(void)
{
    while(USART_kbhit())
    {
        char c = USART_getchar();

        switch(st)
        {
        case ST_IDLE:
            if(c == '&')
            {
                pos = 0;
                frame[pos++] = c;
                st = ST_COLLECT;
            }
            break;

        case ST_COLLECT:
            /* nowy start resetuje ramkę */
            if(c == '&')
            {
                pos = 0;
                frame[pos++] = c;
                break;
            }

            /* zapisuj znak */
            if(pos < sizeof(frame)-1)
                frame[pos++] = c;

            /* koniec ramki */
            if(c == '*')
            {
                frame[pos] = 0;   // string end
                validate_frame(frame,pos);
                st = ST_IDLE;
            }
```

```
            break;
        }
    }
}
```

f. Walidacja ramki

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

1. **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).
2. **Walidacja znaków:** Adresy muszą składać się z dozwolonych znaków (funkcja is_addr_char_valid), a pola ID i LEN wyłącznie z cyfr (is_digits_only).
3. **Zgodność Pola Danych:** System sprawdza, czy faktyczna liczba odebranych znaków odpowiada wartości zapisanej w polu LEN oraz czy dane (np. kody komend) są numeryczne.
4. **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.

```
void validate_frame(char *f, uint16_t flen)
{
    char src[4], dst[4], id[3], len_str[4];

    /* ====== Sprawdzenie minimalnej długości ramki ====== */
    if(flen < 13) {
        USART_fsend("ERR: TOO SHORT\r\n");
        return;
    }

    /* ====== Parsowanie pól ====== */
    uint16_t pos = 1;  // po '&'
    memcpy(src, &f[pos], 3); src[3]=0; pos+=3;
    memcpy(dst, &f[pos], 3); dst[3]=0; pos+=3;
    memcpy(id,  &f[pos], 2); id[2]=0;  pos+=2;
    memcpy(len_str, &f[pos], 3); len_str[3]=0; pos+=3;

    /* ====== Walidacja znaków SRC/DST ====== */
    for (uint8_t i = 0; i < 3; i++) {
        if (!is_addr_char_valid(src[i]) || !is_addr_char_valid(dst[i])) {
            USART_fsend("ERR: INVALID ADDR\r\n");
            return;
        }
    }

    /* ====== Sprawdzenie formatu ID (tylko cyfry 0-9) ====== */
    if(!is_digits_only(id, 2)) {
        USART_fsend("ERR: INVALID ID\r\n");
        return;
    }

    /* ====== Sprawdzenie formatu długości (tylko cyfry 0-9) ====== */
    if(!is_digits_only(len_str, 3)) {
        USART_fsend("ERR\r\n");
        return;
    }

    uint16_t data_len_declared = atoi(len_str);

    /* ====== 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"); //
ERR_NO_DATA
33.                     return;
34.             }
35.             measurement_entry_t *entry = Measurement_GetEntry(count - 1);
36.             if (!entry) {
37.                     send_response_frame(device_addr, src_addr, id, "03"); //
ERR_NO_DATA
38.                     return;
39.             }
40.             uint32_t lux_val = (uint32_t)(entry->lux + 0.5f);
41.             if (lux_val > 9999) {
42.                     lux_val = 9999;
43.             }
44.             char response[5];
45.             response[0] = '0' + (lux_val / 1000) % 10;
46.             response[1] = '0' + (lux_val / 100) % 10;
47.             response[2] = '0' + (lux_val / 10) % 10;
48.             response[3] = '0' + lux_val % 10;
49.             response[4] = 0;
50.             send_response_frame(device_addr, src_addr, id, response);
51.     }
52.     // 13 - VIEW (+ xxzz parametry)
53.     else if (cmd_code == 13) {
```

```
 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: xxLLLLLLLLLLL... (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.                                 }
101.                                 // Dodanie 4-cyfrowej wartości lux
102.                                 data_out[data_pos++] = '0' + (lux_val / 1000) % 10;
103.                                 data_out[data_pos++] = '0' + (lux_val / 100) % 10;
104.                                 data_out[data_pos++] = '0' + (lux_val / 10) % 10;
105.                                 data_out[data_pos++] = '0' + lux_val % 10;
106.                                 batch_size++;
107.                                 measurements_sent++;
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.         }
```

```
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

Konfiguracjia 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

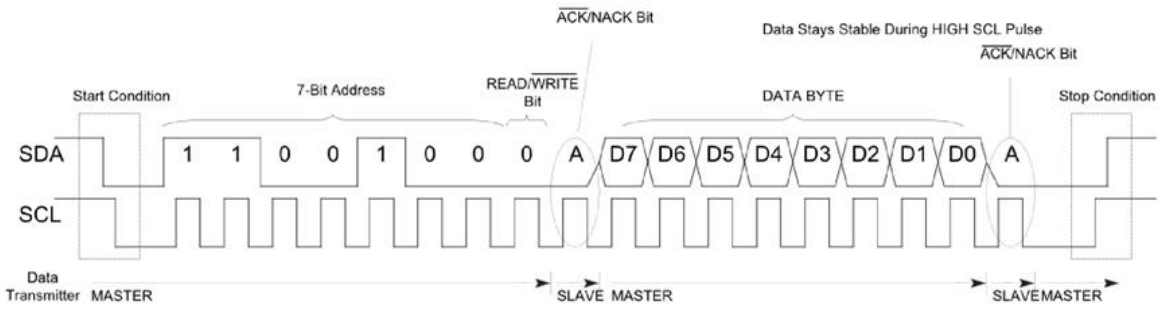
- Clock Source – Internal Clock
- Prescaler – 8999
- Counter Mode – Up
- Counter Period (ARR) – 9
- Clock Division – No Division
- Auto-Reload Preload – Enable
- Częstotliwość przerwań – 1 kHz (1 ms)

4. Opis działania czujnika BH1750

1. Interfejs komunikacyjny czujnika BH1750

VCC	**3.3V**
GND	**GND**
SCL	**PB6**
SDA	**PB7**
ADDR	**0x23 lub 0x5C**

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



2. Konfiguracja początkowa systemu

1. Struktura danych i definicje

```c
// Adresy I2C czujnika BH1750 (7-bit)
#define BH1750_ADDR_LOW    0x23  // ADDR pin do GND
#define BH1750_ADDR_HIGH   0x5C  // ADDR pin do VCC

// Komendy czujnika BH1750
#define BH1750_CONTINUOUS_HIGH_RES_MODE    0x10  // Ciągły wysokiej rozdzielczości (1lx, 120ms)
#define BH1750_CONTINUOUS_HIGH_RES_MODE_2   0x11  // Ciągły wysokiej rozdzielczości 2 (0.5lx, 120ms)
#define BH1750_CONTINUOUS_LOW_RES_MODE      0x13  // Ciągły niskiej rozdzielczości (4lx, 16ms)

// Zmienne globalne
static uint8_t bh1750_current_mode = BH1750_CONTINUOUS_HIGH_RES_MODE;
static uint8_t bh1750_initialized = 0;
static uint8_t bh1750_addr = BH1750_ADDR_LOW;

// Stany maszyny stanów inicjalizacji
typedef enum {
    BH1750_INIT_IDLE = 0,
    BH1750_INIT_PWRON,
    BH1750_INIT_RESET,
    BH1750_INIT_MODE,
    BH1750_INIT_DONE
} bh1750_init_state_t;

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:

```c
// Struktura dla operacji I2C
typedef struct {
    uint8_t address;     // Adres urządzenia I2C (7-bit)
    uint8_t *data;       // Wskaźnik na bufor danych
    uint16_t len;        // Długość danych
    uint8_t operation;   // 0 = TX (transmisja), 1 = RX (odbiór)
    uint8_t pending;     // Flaga: 1 = operacja w toku, 0 = wolne
} i2c_operation_t;

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:

```c
// Timer aplikacyjny oparty o TIM2 (1ms)
static __IO uint32_t app_tick = 0;

static uint32_t App_GetTick(void) {
    return app_tick;
}

// Callback przerwania timera (wywoływany co 1ms)
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
```

```
    if (htim->Instance == TIM2) {
        app_tick++;
    }
}
```

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:

```
// Struktura dla śledzenia czasu oczekiwania BH1750
typedef struct {
    uint32_t start_time;  // Czas rozpoczęcia (App_GetTick())
    uint32_t wait_time;   // Wymagany czas oczekiwania w ms
    uint8_t active;       // 1 = aktywne oczekiwanie, 0 = zakończone
} bh1750_timing_t;

static bh1750_timing_t bh1750_timing = {0};

void BH1750_StartTiming(uint32_t wait_time_ms) {
    bh1750_timing.start_time = App_GetTick();
    bh1750_timing.wait_time = wait_time_ms;
    bh1750_timing.active = 1;
}

uint8_t BH1750_IsTimingReady(void) {
    if (!bh1750_timing.active) {
        return 1; // Nie ma aktywnego oczekiwania
    }

    uint32_t current_time = App_GetTick();
    if ((current_time - bh1750_timing.start_time) >= bh1750_timing.wait_time) {
        bh1750_timing.active = 0;
        return 1; // Czas minął
    }

    return 0; // Jeszcze czeka
}
```

3. Inicjalizacja czujnika przez maszyne stanów

1. *Nieblokująca transmisja I2C*

Funkcja `I2C_Transmit_IT()` inicjuje transmisję przez przerwania:

```
HAL_StatusTypeDef I2C_Transmit_IT(uint8_t address, uint8_t *data, uint16_t len) {
    if (i2c_op.pending) {
        return HAL_BUSY; // Magistrala zajęta
    }

    if (len > I2C_TXBUF_LEN) {
        return HAL_ERROR; // Zbyt duże dane
    }

    // Sprawdź stan HAL I2C
    HAL_I2C_StateTypeDef state = HAL_I2C_GetState(&hi2c1);
    if (state != HAL_I2C_STATE_READY) {
        HAL_I2C_DeInit(&hi2c1);
        HAL_I2C_Init(&hi2c1);
        return HAL_ERROR;
    }

    // Kopiowanie danych do bufora
    for (uint16_t i = 0; i < len; i++) {
        I2C_TxBuf[i] = data[i];
    }

    i2c_op.address = address;
    i2c_op.data = I2C_TxBuf;
```

```
    i2c_op.len = len;
    i2c_op.operation = 0; // TX
    i2c_op.pending = 1;   // Zaznacz zajętość

    // Rozpoczęcie transmisji przez przerwania (adres przesunięty o 1 bit w lewo)
    HAL_StatusTypeDef status = HAL_I2C_Master_Transmit_IT(&hi2c1, address << 1, I2C_TxBuf,
len);

    if (status != HAL_OK) {
        i2c_op.pending = 0; // Zwolnij w przypadku błędu
    }

    return status;
}
```

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:

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

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:

```
void BH1750_Init_Process(void) {
    static uint32_t last_retry_time = 0;

    if (bh1750_initialized) {
        return; // Już zainicjalizowany
    }

    // Inicjuj tylko gdy automatyczny pomiar jest włączony
    if (!measurement_auto.enabled) {
        return;
    }

    // Jeśli był błąd I2C, zresetuj stan inicjalizacji
    if (I2C_Error) {
        I2C_Error = 0;
        bh1750_initialized = 0;
        bh1750_init_state = BH1750_INIT_IDLE;
        return;
    }

    // Nie rozpoczynaj kolejnego kroku, jeśli I2C jest zajęte
    if (i2c_op.pending) {
        return;
    }

    // Throttling - nie próbuj zbyt często (1 sekunda między próbami)
    if (bh1750_init_state == BH1750_INIT_IDLE) {
        uint32_t now = App_GetTick();
        if (now - last_retry_time < 1000) {
            return; // Czekaj
        }
        last_retry_time = now;
    }

    switch (bh1750_init_state) {
```

```
        case BH1750_INIT_IDLE: {
            // Krok 1: Wysłanie komendy POWER_ON (0x01)
            uint8_t cmd = 0x01;
            HAL_StatusTypeDef status = I2C_Transmit_IT(bh1750_addr, &cmd, 1);
            if (status == HAL_OK) {
                bh1750_init_state = BH1750_INIT_PWRON;
            }
            break;
        }
        case BH1750_INIT_PWRON: {
            // Krok 2: Wysłanie komendy RESET (0x07)
            uint8_t cmd = 0x07;
            if (I2C_Transmit_IT(bh1750_addr, &cmd, 1) == HAL_OK) {
                bh1750_init_state = BH1750_INIT_RESET;
            }
            break;
        }
        case BH1750_INIT_RESET: {
            // Krok 3: Ustawienie trybu pomiaru
            if (BH1750_SetMode(bh1750_current_mode) == HAL_OK) {
                bh1750_init_state = BH1750_INIT_MODE;
            }
            break;
        }
        case BH1750_INIT_MODE:
            // Krok 4: Czekanie na zakończenie transmisji trybu
            if (!i2c_op.pending) {
                bh1750_initialized = 1;
                bh1750_init_state = BH1750_INIT_DONE;
            }
            break;
        default:
            break;
    }
}
```

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:

```
HAL_StatusTypeDef BH1750_SetMode(uint8_t mode) {
    HAL_StatusTypeDef status;
    uint8_t addr = bh1750_addr;

    // Wysyłanie komendy trybu do czujnika przez przerwania
    status = I2C_Transmit_IT(addr, &mode, 1);

    if (status == HAL_OK) {
        bh1750_current_mode = mode;

        // Rozpoczęcie śledzenia czasu oczekiwania
        // Opóźnienie w zależności od trybu (120ms dla high res, 16ms dla low res)
        if (mode == BH1750_CONTINUOUS_LOW_RES_MODE || mode == BH1750_ONETIME_LOW_RES_MODE) {
            BH1750_StartTiming(16);
        } else {
            BH1750_StartTiming(120);
        }
    }

    return status;
}
```

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
```

```
            HAL_StatusTypeDef status = BH1750_ReadLight(&bh1750_last_lux);

            // Jeśli I2C zawiódł, dodaj wartość 0 jako wskaźnik błędu
            if (status != HAL_OK) {
                Measurement_AddEntry(0.0f); // Błąd I2C
            }
        }
    }
}
```

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:

```
// Bufory dla odczytu BH1750
static uint8_t bh1750_read_buffer[2] = {0};
static float bh1750_last_lux = 0.0f;
static uint8_t bh1750_read_ready = 0;

HAL_StatusTypeDef BH1750_ReadLight(float *lux) {
    uint8_t addr = bh1750_addr;

    // Sprawdzenie czy operacja I2C nie jest w toku
    if (i2c_op.pending) {
        return HAL_BUSY;
    }

    // Sprawdzenie czy ostatni odczyt jest gotowy
    if (bh1750_read_ready) {
        *lux = bh1750_last_lux;
        bh1750_read_ready = 0;
        return HAL_OK;
    }

    // Rozpoczęcie odczytu przez przerwania
    HAL_StatusTypeDef status = I2C_Receive_IT(addr, bh1750_read_buffer, 2);

    return status; // Wartość zostanie przetworzona w callbacku
}

```

4. *Nieblokujący odbiór I2C*

Funkcja `I2C_Receive_IT()` uruchamia odbiór danych:

```
HAL_StatusTypeDef I2C_Receive_IT(uint8_t address, uint8_t *data, uint16_t len) {
    if (i2c_op.pending) {
        return HAL_BUSY; // Operacja w toku
    }

    if (len > I2C_RXBUF_LEN) {
        return HAL_ERROR; // Zbyt duże dane
    }

    i2c_op.address = address;
    i2c_op.data = data;          // Wskaźnik na bh1750_read_buffer
    i2c_op.len = len;            // 2 bajty
    i2c_op.operation = 1;        // RX (odbiór)
    i2c_op.pending = 1;          // Zajmij magistralę

    // Rozpoczęcie odbioru przez przerwania
    // (address << 1) | 0x01 - adres z bitem odczytu
    HAL_StatusTypeDef status = HAL_I2C_Master_Receive_IT(&hi2c1, (address << 1) | 0x01,
I2C_RxBuf, len);

```

```
    if (status != HAL_OK) {
        i2c_op.pending = 0; // Zwolnij w przypadku błędu
    }

    return status;
}
```

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:

```
void HAL_I2C_MasterRxCpltCallback(I2C_HandleTypeDef *hi2c) {
    if (hi2c == &hi2c1) {
        // Kopiowanie danych z bufora do miejsca docelowego
        if (i2c_op.operation == 1 && i2c_op.data != NULL) {
            for (uint16_t i = 0; i < i2c_op.len; i++) {
                i2c_op.data[i] = I2C_RxBuf[i];
            }

            // Jeśli to odczyt BH1750 (2 bajty)
            if (i2c_op.len == 2 && i2c_op.data == bh1750_read_buffer) {
                // Konwersja bajtów na 16-bitową wartość (big-endian)
                uint16_t raw_value = (bh1750_read_buffer[0] << 8) | bh1750_read_buffer[1];

                // Przeliczenie na luksy: wartość / 1.2 (dla trybu H_RES_MODE)
                bh1750_last_lux = raw_value / 1.2f;

                // Ustaw flagę gotowości
                bh1750_read_ready = 1;
            }
        }
        i2c_op.pending = 0; // Zwolnij magistralę
    }
}
```

Callback wykrywa że to odczyt z BH1750 (2 bajty), łączy oba bajty w wartość 16-bitową, dzieli przez 1.2 i ustawia flagę gotowości. 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:

```
#define MEASUREMENT_BUFFER_SIZE 1000
static measurement_entry_t measurement_buffer[MEASUREMENT_BUFFER_SIZE];
static uint16_t measurement_write_index = 0;  // Indeks do zapisu (head)
static uint16_t measurement_count = 0;        // Liczba zapisanych pomiarów

typedef struct measurement_entry_t {
    float lux;           // Wartość natężenia światła w luksach
    uint32_t timestamp;  // Timestamp pomiaru (App_GetTick())
} measurement_entry_t;

void Measurement_AddEntry(float lux) {
    // Zapisanie pomiaru do bufora
    measurement_buffer[measurement_write_index].lux = lux;
    measurement_buffer[measurement_write_index].timestamp = App_GetTick();

    // Aktualizacja indeksu (bufor cykliczny)
    measurement_write_index++;
    if (measurement_write_index >= MEASUREMENT_BUFFER_SIZE) {
        measurement_write_index = 0; // Zawinięcie bufora
    }

    // 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.