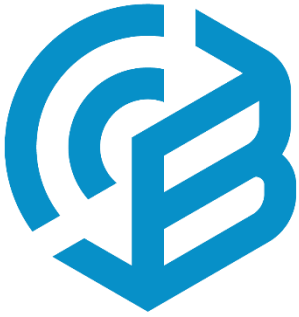


POLITECHNIKA BYDGOSKA
Wydział Telekomunikacji, Informatyki i Elektrotechniki

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

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
0x26	**0x21–0x7E**	**0x21–0x7E**	**0x30-0x39**	**0x30–0x39**	**0x30–0x39**	**0x30–0x39, 0x41-0x46** oraz **0x61–0x66**	**0x2A**
&	Wszystkie znaki ASCII poza **0x26 i 0x2a**	Wszystkie znaki ASCII poza **0x26 i 0x2a**	Cyfry ASCII 0-9	Cyfry ASCII 0-9 w systemie dziesiętnym	Cyfry ASCII 0-9 w systemie dziesiętnym	Znaki w systemie ASCII od **0** do **9**, **A** do **F** oraz **a** do **f**	*
1 znak	3 znaki	3 znaki	2 znaki	3 znaki	0-256 znaków	2 znaki	1 znak
Znak startu	3-znakowy identyfikator nadawcy	3-znakowy identyfikator odbiorcy	2-znakowy indetyfikator ramki	Liczba znaków w polu Dane po dekodowaniu (max 256). Nadawca musi walidować, aby nie przekroczyć limitu	Pole danych, może być puste.	Obliczana po dekodowaniu pola Dane. Używamy CRC-8	Znak końca

Początek ramki

- jednoznacznie oznacza początek nowej ramki
- każdy & poza poprawną ramką powoduje restart parsera

Nadawca

- 3-znakowy identyfikator nadawcy
- wyłącznie znaki ASCII poza **0x26 i 0x2a**

Odbiorca

- 3-znakowy identyfikator odbiorcy
- wyłącznie znaki ASCII poza **0x26 i 0x2a**

Identyfikator ramki

- numer ramki
- wszystkie znaki w systemie dziesiętnym
- zakres 00–99

Długość danych

- liczba znaków w polu DANE

- wszystkie znaki w systemie dziesiętnym
- nie może być większe niż 256

Dane

- cyfry od 0-9
- wszystkie znaki w systemie dziesiętnym
- może być puste (LEN = 000)

Suma kontrolna

- Obliczana przy użyciu CRC-8 z wielomianem x^8 + x^2 + x + 1 (0x07).
- Wynik zapisany jako **2-cyfrowa liczba ASCII**, reprezentowana jako hexadecymalna.

Koniec ramki

- jednoznacznie zamyka ramkę

2. Kontrola poprawności

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

3. Komendy

Kody Wysyłek:

- 10 – START
- 11 – STOP
- 12 – DOWNLOAD
- 13 – VIEW (+ xxxzzz)
- 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 odczytywać dane z czujnika BH1750 w określonych odstępach czasu.
- Proces pomiaru i zapisu do bufora odbywa się automatycznie

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 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	005	02600	62	*

4. Podgląd wpisów (13xxxzzz)

- Szybki podgląd ostatnich N wpisów (bieżących) i opcjonalnie starszych wpisów (archiwum).
- Domyślnie wysyła ostatnie pomiary; opcjonalny parametr w polu Dane umożliwia wybranie starszych wpisów.
- xxx –offset – najnowszy wpis z kolei np. ‘000’ to najnowszy wpis, a ‘999’ najstarszy
- zzz – ilość wpisów od offsetu, które ma wyświetlić
- Jeżeli ktoś chce pobrać wszystkie 1000 wpisów to wystarczy wpisać w dane ‘000000’

Wysyłka

- 13 = kod komendy VIEW
- 000 = ID początkowego wpisu (najstarszy pomiar)

- 100 = pobierz 100 pomiarów

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	PC_	STM	04	008	13000100	2E	*

Odbiór

Jeśli żądana liczba pomiarów przekracza *MAX_MEASUREMENTS_PER_FRAME* (czyli 50), dane są dzielone na kilka ramek. Każda ramka zawiera:

- 3 cyfry offsetu (oznaczające pozycję pierwszego pomiaru w tej ramce względem najnowszego wpisu),
- do 50 kolejnych pomiarów (każdy po 5 cyfr),
- jeśli żądanych pomiarów jest więcej, kolejne ramki będą miały zwiększony offset (np. 000, 050, 100,

Jeśli żądasz np. 100 pomiarów system wyśle 2 ramki:

Ramka 1

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	STM	PC_	04	253	00002600025…	DC	*

Ramka 2

Początek ramki	Nadawca	Odbiorca	ID	Długość danych	Dane	Suma kontrolna	Koniec ramki
&	STM	PC_	04	253	05002550025…	8C	*

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 trybu 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 trybu 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;

// Struktura pojedynczego pomiaru
typedef struct {
    float lux;
    uint32_t timestamp;
} measurement_entry_t;

// Struktura automatycznego pomiaru
typedef struct {
    uint32_t interval_ms;
    uint32_t last_measurement;
    uint8_t enabled;
} measurement_auto_t;
 // Stała określająca maksymalną liczbę pomiarów wysyłanych w jednej ramce.
#define MAX_MEASUREMENTS_PER_FRAME 50
```

b. Zmienne globalne

```
// Obsługa UART (USART)
extern UART_HandleTypeDef huart2;      // Uchwyt do peryferium UART2 (do komunikacji)
extern uint8_t rx_byte;                // Ostatni odebrany bajt przez UART

/* Bufory USART */
uint8_t USART_TxBuf[USART_TXBUF_LEN];  // Bufor danych do wysłania przez UART
uint8_t USART_RxBuf[USART_RXBUF_LEN];  // Bufor danych odebranych przez UART

__IO int USART_TX_Empty;               // Wskaźnik pozycji pustej w buforze TX (head)
__IO int USART_TX_Busy;                // Wskaźnik pozycji zajętej w buforze TX (tail)
__IO int USART_RX_Empty;               // Wskaźnik pozycji pustej w buforze RX (head)
__IO int USART_RX_Busy;                // Wskaźnik pozycji zajętej w buforze RX (tail)

__IO uint8_t USART_RxBufOverflow;      // Flaga przepełnienia bufora odbiorczego UART

/* Bufor pomiarów światła */
static measurement_entry_t LightBuffer[LIGHT_BUFFER_SIZE]; // Cykliczny bufor pomiarów światła (struktury measurement_entry_t)
static volatile uint32_t LightBuffer_WritePos;             // Pozycja zapisu w buforze pomiarów (indeks cykliczny)
```

```
19. static volatile uint32_t LightBuffer_Count;                // Liczba pomiarów aktualnie
zapisanych w buforze
20. static volatile uint8_t LightBuffer_DataAvailable;         // Flaga informująca o dostępności
nowych danych w buforze
```

c. Funkcje komunikacji 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().

```
 1. // Sprawdzenie, czy w buforze odbiorczym USART są nowe dane do odczytu
 2. uint8_t USART_kbhit(void) {
 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. /**
11.  * @brief Pobiera jeden znak z bufora odbiorczego USART
12.  * @retval Znak z bufora lub -1 jeśli bufor jest pusty
13.  */
14. int16_t USART_getchar(void) {
15.     int16_t tmp;
16.     // Sprawdzenie, czy w buforze odbiorczym są dane
17.     if (USART_RX_Empty != USART_RX_Busy) {
18.         tmp = USART_RxBuf[USART_RX_Busy]; // Pobranie znaku z bufora
19.         USART_RX_Busy++;
20.         if (USART_RX_Busy >= USART_RXBUF_LEN)
21.             USART_RX_Busy = 0; // Bufor cykliczny
22.         return tmp;
23.     } else
24.         return -1; // Bufor pusty
25. }
26.
27. /**
28.  * @brief Wysyłanie sformatowanego tekstu przez UART
29.  * @param format: łańcuch formatujący (jak printf)
30.  */
31. void USART_fsend(char *format, ...) {
32.     char tmp_rs[560]; // Bufor na sformatowany tekst
33.     int i;
34.     __IO int idx;
35.     va_list arglist;
36.     va_start(arglist, format);
37.     vsprintf(tmp_rs, format, arglist); // Formatowanie tekstu
38.     va_end(arglist);
39.     idx = USART_TX_Empty;
40.     // Przepisanie sformatowanego tekstu do bufora wysyłki
41.     for (i = 0; i < strlen(tmp_rs); i++) {
42.         USART_TxBuf[idx] = tmp_rs[i];
43.         idx++;
44.         if (idx >= USART_TXBUF_LEN)
```

```
45.             idx = 0; // Bufor cykliczny
46.     }
47.     __disable_irq(); // Wyłączenie przerwań na czas modyfikacji wskaźników
48.     // Jeśli bufor jest pusty i UART gotowy do wysyłki, rozpocznij transmisję
49.     if ((USART_TX_Empty == USART_TX_Busy)
50.             && (__HAL_UART_GET_FLAG(&huart2, UART_FLAG_TXE) == SET)) {
51.         USART_TX_Empty = idx;
52.         uint8_t tmp = USART_TxBuf[USART_TX_Busy];
53.         USART_TX_Busy++;
54.         if (USART_TX_Busy >= USART_TXBUF_LEN)
55.             USART_TX_Busy = 0;
56.         HAL_UART_Transmit_IT(&huart2, &tmp, 1); // Rozpocznij wysyłanie przez przerwanie
57.     } else {
58.         USART_TX_Empty = idx;
59.     }
60.     __enable_irq(); // Włączenie przerwań
61. }
```

d. 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

```
1. void process_uart_buffer(void) {
2.     while (USART_kbhit()) {
3.         char c = USART_getchar();
4.
5.         switch (st) {
6.             case ST_IDLE: {
7.                 if (c == '&') {
8.                     pos = 0;
9.                     frame[pos++] = c;
10.                    st = ST_COLLECT;
11.                }
12.                break;
13.            }
14.
15.            case ST_COLLECT: {
16.                if (c == '&') {
17.                    pos = 0;
18.                    frame[pos++] = c;
19.                    break;
20.                }
21.
22.                if (pos < sizeof(frame) - 1) {
23.                    frame[pos++] = c;
24.                } else {
25.                    // Przepełnienie bufora: odrzucenie ramki i powrót do ST_IDLE
26.                    pos = 0;
27.                    st = ST_IDLE;
28.                    break;
29.                }
30.
31.                if (c == '*') {
```

```
                    frame[pos] = 0; // Terminator stringa
                    validate_frame(frame, pos); // Walidacja ramki
                    st = ST_IDLE;
                }
                break;
            }
        }
    }
}
```

e. Walidacja ramki

Walidacja ramki polega na sprawdzeniu poprawności strukturalnej i logicznej odebranej ramki:

Sprawdzana jest długość ramki, poprawność pól adresowych, ID, LEN, CRC oraz zgodność deklarowanej długości danych z rzeczywistą.
W przypadku błędów wysyłany jest odpowiedni komunikat dla developera
(np. ERR: TOO SHORT, ERR: INVALID ADDR, ERR_LENGTH, ERR_CRC).
Jeśli ramka jest poprawna, przekazywana jest do dalszej obsługi (handle_command).

```
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 ====== */
    if(data_len_declared > 256) {
        USART_fsend("ERR_LENGTH\r\n");
        return;
    }

    /* ====== 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. }
```

f. Obsługa ramki

Obsługa ramki polega na analizie pola DANE i wykonaniu odpowiedniej akcji w zależności od kodu komendy:

Rozpoznawane są komendy: START(10), STOP(11), DOWNLOAD(12), VIEW(13), SET_INTERVAL(14), GET_INTERVAL(15), SET_MODE(16), GET_MODE(17).
W przypadku błędnych parametrów lub braku danych wysyłane są odpowiednie kody błędów.
Wyniki lub status operacji są zwracane w odpowiedzi przez *send_response_frame*.

```
 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.     const uint16_t MIN_INTERVAL = 1;
 5.     const uint16_t MAX_INTERVAL = 9999;
 6.
 7.     if (strlen(cmd) < 2 || !is_digits_only(cmd, 2)) {
 8.         USART_fsend("ERR: INVALID CMD\r\n");
 9.         return;
10.     }
11.
12.     uint8_t cmd_code = (cmd[0] - '0') * 10 + (cmd[1] - '0');
13.     const char *params = (strlen(cmd) > 2) ? &cmd[2] : "";
14.
15.     // 10 - START
16.     if (cmd_code == 10) {
17.         Measurement_EnableAutoRead(1);
18.         send_response_frame(device_addr, src_addr, id, "00");
19.     }
20.     // 11 - STOP
21.     else if (cmd_code == 11) {
22.         Measurement_EnableAutoRead(0);
23.         send_response_frame(device_addr, src_addr, id, "00");
24.     }
25.     // 12 - DOWNLOAD (ostatni pomiar)
26.     else if (cmd_code == 12) {
27.         uint16_t count = (uint16_t)LightBuffer_GetCount();
28.         if (count == 0) {
29.             send_response_frame(device_addr, src_addr, id, "03");
30.             return;
31.         }
32.         measurement_entry_t *entry = LightBuffer_GetLatest();
33.
34.         uint32_t lux_val = (uint32_t)(entry->lux + 0.5f);
35.         if (lux_val > 65535) {
36.             lux_val = 65535;
37.         }
38.         char response[6];
39.         response[0] = '0' + (lux_val / 10000) % 10;
40.         response[1] = '0' + (lux_val / 1000) % 10;
41.         response[2] = '0' + (lux_val / 100) % 10;
42.         response[3] = '0' + (lux_val / 10) % 10;
43.         response[4] = '0' + lux_val % 10;
44.         response[5] = 0;
45.         send_response_frame(device_addr, src_addr, id, response);
46.     }
47.     // 13 - VIEW (+ xxxzzz parametry)
48.     else if (cmd_code == 13) {
49.         if (strlen(params) < 6) {
50.             send_response_frame(device_addr, src_addr, id, "01");
51.             return;
52.         }
53.         uint16_t start_offset = (params[0] - '0') * 100 + (params[1] - '0') * 10 +
(params[2] - '0');
54.         uint16_t count_req = (params[3] - '0') * 100 + (params[4] - '0') * 10 + (params[5] -
'0');
```

```
        uint16_t count = (uint16_t)LightBuffer_GetCount();
        if (start_offset >= count) {
            send_response_frame(device_addr, src_addr, id, "03");
            return;
        }
        // pobranie wszystkich danych z bufora dla '000000'
        if (start_offset == 0 && count_req == 0) {
            count_req = count;
        } else if (count_req == 0) {
            count_req = count - start_offset;
        }

        #define MAX_MEASUREMENTS_PER_FRAME 50
        char data_out[256];
        uint16_t measurements_sent = 0;

        while (measurements_sent < count_req) {
            uint16_t current_offset = start_offset + measurements_sent;
            uint16_t batch_size = 0;
            uint16_t data_pos = 0;

            data_out[data_pos++] = '0' + (current_offset / 100) % 10;
            data_out[data_pos++] = '0' + (current_offset / 10) % 10;
            data_out[data_pos++] = '0' + current_offset % 10;

            for (uint16_t i = 0; i < MAX_MEASUREMENTS_PER_FRAME && measurements_sent <
count_req; i++) {
                int32_t idx = (int32_t)count - 1 - (int32_t)current_offset - (int32_t)i;
                if (idx < 0) {
                    break;
                }
                measurement_entry_t *entry = LightBuffer_GetByIndexOldest((uint16_t)idx);
                if (!entry) {
                    break;
                }
                uint32_t lux_val = (uint32_t)(entry->lux + 0.5f);
                if (lux_val > 65535) {
                    lux_val = 65535;
                }
                data_out[data_pos++] = '0' + (lux_val / 10000) % 10;
                data_out[data_pos++] = '0' + (lux_val / 1000) % 10;
                data_out[data_pos++] = '0' + (lux_val / 100) % 10;
                data_out[data_pos++] = '0' + (lux_val / 10) % 10;
                data_out[data_pos++] = '0' + lux_val % 10;
                batch_size++;
                measurements_sent++;
            }

            data_out[data_pos] = 0;
            send_response_frame(device_addr, src_addr, id, data_out);

            if (batch_size == 0) {
                break;
            }
        }
    }
    // 14 - SET_INTERVAL (+ xxxx parametr)
    else if (cmd_code == 14) {
        if (strlen(params) < 4) {
            send_response_frame(device_addr, src_addr, id, "01");
            return;
        }
        uint16_t interval_ms = atoi(params);
        if (interval_ms < MIN_INTERVAL || interval_ms > MAX_INTERVAL) {
            send_response_frame(device_addr, src_addr, id, "02");
            return;
        }
        Measurement_SetInterval(interval_ms);
        send_response_frame(device_addr, src_addr, id, "00");
```

```
    }
    // 15 - GET_INTERVAL
    else if (cmd_code == 15) {
        uint32_t interval_ms = Measurement_GetInterval();
        if (interval_ms > 9999) {
            interval_ms = 9999;
        }
        char response[6];
        response[0] = '0' + (interval_ms / 1000) % 10;
        response[1] = '0' + (interval_ms / 100) % 10;
        response[2] = '0' + (interval_ms / 10) % 10;
        response[3] = '0' + interval_ms % 10;
        response[4] = 0;
        send_response_frame(device_addr, src_addr, id, response);
    }
    // 16 - SET_MODE (+ x parametr)
    else if (cmd_code == 16) {
        if (strlen(params) < 1 || params[0] < '1' || params[0] > '6') {
            send_response_frame(device_addr, src_addr, id, "01");
            return;
        }
        uint8_t mode_num = params[0] - '0';
        uint8_t mode_value;
        switch (mode_num) {
            case 1: mode_value = BH1750_CONTINUOUS_HIGH_RES_MODE; break;
            case 2: mode_value = BH1750_CONTINUOUS_HIGH_RES_MODE_2; break;
            case 3: mode_value = BH1750_CONTINUOUS_LOW_RES_MODE; break;
            case 4: mode_value = BH1750_ONETIME_HIGH_RES_MODE; break;
            case 5: mode_value = BH1750_ONETIME_HIGH_RES_MODE_2; break;
            case 6: mode_value = BH1750_ONETIME_LOW_RES_MODE; break;
        }
        HAL_StatusTypeDef status = BH1750_SetMode(mode_value);
        if (status == HAL_OK) {
            send_response_frame(device_addr, src_addr, id, "00");
        } else {
            send_response_frame(device_addr, src_addr, id, "04");
        }
    }
    // 17 - GET_MODE
    else if (cmd_code == 17) {
        char mode_char = '1';
        uint8_t current_mode = BH1750_GetCurrentMode();
        switch (current_mode) {
            case BH1750_CONTINUOUS_HIGH_RES_MODE: mode_char = '1'; break;
            case BH1750_CONTINUOUS_HIGH_RES_MODE_2: mode_char = '2'; break;
            case BH1750_CONTINUOUS_LOW_RES_MODE: mode_char = '3'; break;
            case BH1750_ONETIME_HIGH_RES_MODE: mode_char = '4'; break;
            case BH1750_ONETIME_HIGH_RES_MODE_2: mode_char = '5'; break;
            case BH1750_ONETIME_LOW_RES_MODE: mode_char = '6'; break;
            default: mode_char = '1'; break;
        }
        char response[2];
        response[0] = mode_char;
        response[1] = 0;
        send_response_frame(device_addr, src_addr, id, response);
    }
    else {
        USART_fsend("ERR: UNKNOWN CMD\r\n");
    }
}
```

g. Funkcje pomocnicze

Funkcje pomocnicze wspierają obsługę protokołu komunikacyjnego – sprawdzają poprawność znaków, konwertują dane, obliczają sumę kontrolną.

```
/**
 * @brief Sprawdza, czy podany znak jest cyfrą
 * @param str: wskaźnik na string
 * @param len: długość stringa
```

```
 * @retval 1 jeśli wszystkie znaki są cyframi, 0 w przeciwnym razie
 */
uint8_t is_digits_only(const char *str, uint16_t len) {
    for (uint16_t i = 0; i < len; i++) {
        if (str[i] < '0' || str[i] > '9') {
            return 0;
        }
    }
    return 1;
}

/**
 * @brief Sprawdza, czy znak jest poprawny dla pola adresowego
 * @param c: znak do sprawdzenia
 * @retval 1 jeśli znak jest poprawny, 0 w przeciwnym razie
 */
uint8_t is_addr_char_valid(char c) {
    return (c >= 0x21 && c <= 0x7E && c != '&' && c != '*');
}

/**
 * @brief Sprawdza, czy znak jest cyfrą szesnastkową
 * @param c: znak do sprawdzenia
 * @retval 1 jeśli znak jest hex, 0 w przeciwnym razie
 */
uint8_t is_hex_char(char c) {
    return ((c >= '0' && c <= '9') || (c >= 'A' && c <= 'F'));
}

/**
 * @brief Konwertuje dwa znaki hex na bajt
 * @param c1, c2: znaki hex
 * @retval wartość bajtu
 */
uint8_t hex2byte(char c1, char c2) {
    uint8_t hi = (c1 >= 'A') ? (c1 - 'A' + 10) : (c1 - '0');
    uint8_t lo = (c2 >= 'A') ? (c2 - 'A' + 10) : (c2 - '0');
    return (hi << 4) | lo;
}

/**
 * @brief Konwertuje bajt na dwa znaki hex
 * @param b: bajt
 * @param hex: wskaźnik na bufor wyjściowy
 */
void byte2hex(uint8_t b, char *hex) {
    hex[0] = "0123456789ABCDEF"[b >> 4];
    hex[1] = "0123456789ABCDEF"[b & 0x0F];
}

/**
 * @brief Oblicza sumę kontrolną CRC-8 dla bufora
 * @param buf: wskaźnik na dane
 * @param len: długość danych
 * @retval wartość CRC-8
 */
uint8_t crc8(const uint8_t *buf, uint16_t len) {
    uint8_t crc = 0;
    for (uint16_t i = 0; i < len; i++) {
        crc ^= buf[i];
        for (uint8_t j = 0; j < 8; j++) {
            if (crc & 0x80)
                crc = (crc << 1) ^ 0x07;
            else
                crc <<= 1;
        }
    }
    return crc;
}
```

Funkcja send_response_frame generuje ramkę odpowiedzi zgodną z protokołem, oblicza CRC, formatuje dane i wysyła przez USART.

```
/**
 * @brief Wysyła ramkę odpowiedzi zgodną z protokołem
 * @param src_addr: adres nadawcy
 * @param dst_addr: adres odbiorcy
 * @param id: identyfikator ramki
 * @param data: pole DANE (string)
 */
void send_response_frame(const char *src_addr, const char *dst_addr, const char *id, const char *data) {
    char frame[271];
    uint16_t pos = 0;
    uint8_t crc_buf[300];
    uint16_t crc_pos = 0;

    uint16_t data_len = strlen(data);
    if (data_len > 256) {
        return;
    }

    uint16_t total_len = 1 + 3 + 3 + 2 + 3 + data_len + 2 + 1 + 1;
    if (total_len > sizeof(frame)) {
        // Przekroczenie bufora ramki
        return;
    }

    frame[pos++] = '&';

    memcpy(&frame[pos], src_addr, 3);
    pos += 3;
    memcpy(&crc_buf[crc_pos], src_addr, 3);
    crc_pos += 3;

    memcpy(&frame[pos], dst_addr, 3);
    pos += 3;
    memcpy(&crc_buf[crc_pos], dst_addr, 3);
    crc_pos += 3;

    memcpy(&frame[pos], id, 2);
    pos += 2;
    memcpy(&crc_buf[crc_pos], id, 2);
    crc_pos += 2;

    char len_str[4];
    len_str[0] = '0' + (data_len / 100) % 10;
    len_str[1] = '0' + (data_len / 10) % 10;
    len_str[2] = '0' + data_len % 10;
    len_str[3] = 0;
    memcpy(&frame[pos], len_str, 3);
    pos += 3;
    memcpy(&crc_buf[crc_pos], len_str, 3);
    crc_pos += 3;

    memcpy(&frame[pos], data, data_len);
    pos += data_len;
    memcpy(&crc_buf[crc_pos], data, data_len);
    crc_pos += data_len;

    uint8_t crc = crc8(crc_buf, crc_pos);
    char crc_hex[3];
    byte2hex(crc, crc_hex);
    crc_hex[2] = 0;

    memcpy(&frame[pos], crc_hex, 2);
    pos += 2;

    frame[pos++] = '*';
```

```
66.     frame[pos] = 0;
67.
68.     USART_fsend("%s", frame);
69. }
```

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

4. Opis działania czujnika BH1750

BH1750 to cyfrowy czujnik natężenia światła, umożliwia bezpośredni pomiar natężenia oświetlenia w jednostkach luksów (lx) i komunikuje się z mikrokontrolerem za pomocą interfejsu I2C.

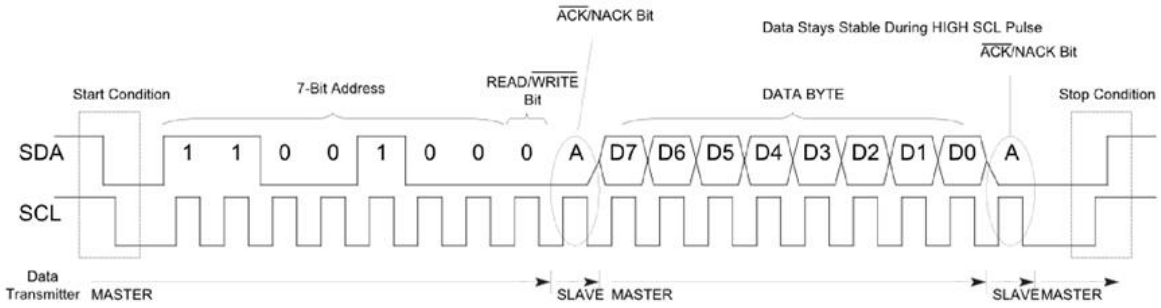
Najważniejsze cechy:

Zakres pomiarowy: od 1 lx do 65535 lx
Rozdzielczość: 1 lx, 0.5 lx lub 4 lx (w zależności od wybranego trybu pracy)
Tryby pracy:
Continuous: czujnik cyklicznie wykonuje pomiary i udostępnia najnowszą wartość
One Time: czujnik wykonuje pojedynczy pomiar po wyzwoleniu, następnie przechodzi w stan uśpienia
Możliwość wyboru adresu I2C (LOW/HIGH) w zależności od podłączenia pinów

1. Opis 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. Definicje typów i stałych

W tej sekcji znajdują się typy wyliczeniowe, struktury oraz stałe związane z obsługą czujnika BH1750. Określają one tryby pracy, stany automatu, konfigurację pomiarów i adres czujnika.

```c
// Tryby pracy czujnika BH1750
#define BH1750_CONTINUOUS_HIGH_RES_MODE    0x10
#define BH1750_CONTINUOUS_HIGH_RES_MODE_2  0x11
#define BH1750_CONTINUOUS_LOW_RES_MODE     0x13
#define BH1750_ONETIME_HIGH_RES_MODE       0x20
#define BH1750_ONETIME_HIGH_RES_MODE_2     0x21
#define BH1750_ONETIME_LOW_RES_MODE        0x23

// Adresy I2C czujnika BH1750
#define BH1750_ADDR_LOW  0x23
#define BH1750_ADDR_HIGH 0x5C

// Typ wyliczeniowy stanów automatu BH1750
typedef enum {
    BH1750_STATE_IDLE = 0,         // Spoczynek
    BH1750_STATE_CONFIGURING,      // Konfiguracja czujnika
    BH1750_STATE_POWERUP_WAIT,     // Oczekiwanie po włączeniu zasilania
    BH1750_STATE_MEASURING,        // Oczekiwanie na zakończenie pomiaru One Time
    BH1750_STATE_READY,            // Czujnik gotowy do odczytu
    BH1750_STATE_BUSY,             // Oczekiwanie na zakończenie transmisji I2C
    BH1750_STATE_ERROR             // Błąd czujnika/I2C
} bh1750_state_t;

// Struktura automatycznego pomiaru
typedef struct {
    uint32_t interval_ms;      // Interwał pomiaru w ms
    uint32_t last_measurement; // Czas ostatniego pomiaru
    uint8_t enabled;           // Flaga włączenia automatycznego odczytu
} measurement_auto_t;
```

3. Zmienne globalne i stanu czujnika

W tej sekcji znajdują się zmienne globalne przechowujące stan czujnika BH1750, konfigurację pomiarów, flagi błędów oraz zmienne pomocnicze do obsługi automatu stanów i pomiarów.

```c
// Uchwyt do I2C (zewnętrzny, inicjalizowany w main.c)
extern I2C_HandleTypeDef hi2c1;

// Flaga błędu I2C (ustawiana przy błędach transmisji)
static __IO uint8_t I2C_Error = 0;

// Aktualny tryb pracy czujnika BH1750
static uint8_t bh1750_current_mode = BH1750_CONTINUOUS_HIGH_RES_MODE;

// Adres I2C czujnika (LOW lub HIGH)
static uint8_t bh1750_addr = BH1750_ADDR_LOW;

// Zmienna przechowująca stan automatu BH1750
static bh1750_state_t bh1750_state = BH1750_STATE_IDLE;

```

```
// Znacznik czasu ostatniej zmiany stanu automatu
static uint32_t bh1750_state_tick = 0;

// Zmienna fazy pomiaru One Time (0=brak, 1=czekanie, 2=gotowy do odczytu)
static uint8_t onetime_meas_phase = 0;

// Znacznik czasu dla timeoutu fazy One Time
static uint32_t onetime_phase_tick = 0;

// Struktura konfiguracji automatycznego pomiaru
static measurement_auto_t measurement_auto = {
    .interval_ms = 1000,
    .last_measurement = 0,
    .enabled = 0
};

// Bufor odczytu danych z czujnika (2 bajty)
static uint8_t bh1750_read_buffer[2] = {0};

// Zmienna pomocnicza do kroków konfiguracji
static uint8_t config_step = 0;

// Komenda Power On dla czujnika
static uint8_t bh1750_power_on_cmd = 0x01;
```

4. Funkcje inicjalizujące i konfigurujące

Funkcje tej sekcji odpowiadają za inicjalizację czujnika BH1750, ustawianie trybu pracy, adresu I2C oraz interwału pomiarowego. Pozwalają przygotować czujnik do pracy w wybranym trybie i z wybraną konfiguracją

```
/**
 * @brief Sprawdza, czy tryb pracy to One Time
 * @param mode: kod trybu BH1750
 * @retval 1 jeśli tryb One Time, 0 jeśli Continuous
 */
static uint8_t is_onetime_mode(uint8_t mode) {
    return (mode == 0x20 || mode == 0x21 || mode == 0x23);
}

/**
 * @brief Ustawia tryb pracy czujnika BH1750 (wysyła komendę przez I2C)
 * @param mode: kod trybu BH1750
 * @retval HAL_OK jeśli sukces, HAL_ERROR jeśli błąd
 */
HAL_StatusTypeDef BH1750_SetMode(uint8_t mode) {
    static uint8_t mode_cmd;
    HAL_StatusTypeDef status;

    mode_cmd = mode;
    status = HAL_I2C_Master_Transmit_IT(&hi2c1, bh1750_addr << 1, &mode_cmd, 1);

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

    return status;
}

/**
 * @brief Ustawia interwał automatycznego pomiaru
 * @param interval_ms: interwał w milisekundach
 */
void Measurement_SetInterval(uint32_t interval_ms) {
    measurement_auto.interval_ms = interval_ms;
}

/**
 * @brief Pobiera aktualny interwał automatycznego pomiaru
 * @retval interwał w milisekundach
```

```
 */
uint32_t Measurement_GetInterval(void) {
    return measurement_auto.interval_ms;
}

/**
 * @brief Włącza lub wyłącza automatyczny odczyt pomiarów
 * @param enable: 1 = włącz, 0 = wyłącz
 */
void Measurement_EnableAutoRead(uint8_t enable) {
    measurement_auto.enabled = enable;
    if (enable) {
        measurement_auto.last_measurement = HAL_GetTick();
    }
}
```

5. Funkcje nieblokujące

Funkcje nieblokujące realizują obsługę czujnika BH1750 w sposób asynchroniczny, bez zatrzymywania głównej pętli programu. Wykorzystują automat stanów, przerwania I2C oraz callbacki, dzięki czemu pomiar i odczyt danych odbywa się w tle.

```
void BH1750_Init_Process(void) {
    if (!measurement_auto.enabled) return;
    if (I2C_Error) {
        // ...obsługa błędu I2C...
        return;
    }
    switch (bh1750_state) {
        case BH1750_STATE_IDLE:
            bh1750_state = BH1750_STATE_CONFIGURING;
            config_step = 0;
            break;
        case BH1750_STATE_CONFIGURING:
            switch (config_step) {
                case 0:
                    // Wyślij komendę Power On
                    // ...kod wysyłający komendę...
                    break;
                case 1:
                    // Tryb Continuous: wyślij komendę trybu w init
                    // Tryb One Time: pomiń
                    // ...kod ustawiający tryb...
                    break;
                case 2:
                    bh1750_state = BH1750_STATE_POWERUP_WAIT;
                    bh1750_state_tick = HAL_GetTick();
                    config_step = 0;
                    break;
                default:
                    // ...obsługa nieoczekiwanej wartości config_step...
                    break;
            }
            break;
        case BH1750_STATE_POWERUP_WAIT:
            // ...oczekiwanie na gotowość po zasileniu...
            break;
        case BH1750_STATE_MEASURING:
            // ...oczekiwanie na zakończenie pomiaru One Time...
            break;
        case BH1750_STATE_READY:
            // Czujnik gotowy, będzie wyzwolony przez Measurement_AutoRead_Process
            break;
        case BH1750_STATE_BUSY:
            // ...timeout na transmisję I2C...
            break;
        case BH1750_STATE_ERROR:
            // ...obsługa błędu i próba ponownej inicjalizacji...
```

```
            break;
        default:
            break;
    }
}
void Measurement_AutoRead_Process(void) {
    if (!measurement_auto.enabled) return;
    // ...obsługa trybu One Time i Continuous, wyzwalanie pomiarów, obsługa timeoutów...
}
/**
 * @brief Callback ukończenia odczytu I2C
 * Wywoływany automatycznie przez HAL po odebraniu danych z czujnika BH1750.
 */
void HAL_I2C_MasterRxCpltCallback(I2C_HandleTypeDef *hi2c) {
    if (hi2c != &hi2c1) {
        return; // Ignoruj, jeśli to nie jest nasz I2C
    }

    // Sprawdzenie, czy oczekiwaliśmy na dane z czujnika
    if (bh1750_state == BH1750_STATE_BUSY) {
        // Połącz dwa bajty z bufora w wartość 16-bitową
        uint16_t raw_value = (bh1750_read_buffer[0] << 8) | bh1750_read_buffer[1];

        // Wybierz dzielnik w zależności od trybu pomiaru
        float lux_divider;
        switch (bh1750_current_mode) {
            case BH1750_CONTINUOUS_HIGH_RES_MODE_2:  // Tryb ciągły H-Res2
            case BH1750_ONETIME_HIGH_RES_MODE_2:     // Tryb jednorazowy H-Res2
                lux_divider = 2.4f;  // Rozdzielczość 0.5 lx
                break;

            case BH1750_CONTINUOUS_LOW_RES_MODE:     // Tryb ciągły L-Res
            case BH1750_ONETIME_LOW_RES_MODE:        // Tryb jednorazowy L-Res
                lux_divider = 0.5f;  // Rozdzielczość 4 lx
                break;

            case BH1750_CONTINUOUS_HIGH_RES_MODE:    // Tryb ciągły H-Res
            case BH1750_ONETIME_HIGH_RES_MODE:       // Tryb jednorazowy H-Res
            default:
                lux_divider = 1.2f;  // Rozdzielczość 1 lx
                break;
        }

        // Przelicz wartość na luks i zapisz do bufora pomiarów
        float lux = raw_value / lux_divider;
        LightBuffer_Put(lux);

        // W trybie One Time wyłącz automatyczny odczyt po jednym pomiarze
        if (is_onetime_mode(bh1750_current_mode)) {
            measurement_auto.enabled = 0;
            onetime_meas_phase = 0;
        }

        // Ustaw stan czujnika na gotowy
        bh1750_state = BH1750_STATE_READY;
    }
}

/**
 * @brief Callback błędu I2C
 * Wywoływany automatycznie przez HAL w przypadku błędu transmisji I2C.
 */
void HAL_I2C_ErrorCallback(I2C_HandleTypeDef *hi2c) {
    if (hi2c == &hi2c1) {
        I2C_Error = 1; // Ustaw flagę błędu do obsługi w automacie stanów
    }
}
```

6. Funkcje pomiarowe i obsługa bufora

Funkcje tej sekcji odpowiadają za zapis pomiarów do bufora cyklicznego, pobieranie najnowszego pomiaru, pobieranie liczby pomiarów oraz dostęp do pomiarów według indeksu lub offsetu. Bufor działa w sposób cykliczny – po zapełnieniu najstarsze pomiary są nadpisywane.

```c
/**
 * @brief Dodaje nowy pomiar do bufora cyklicznego
 * @param lux: Wartość pomiaru światła (w luksach)
 * @retval 1 jeśli pomiar został dodany
 */
uint8_t LightBuffer_Put(float lux) {
    __disable_irq(); // Zabezpieczenie przed przerwaniem w trakcie zapisu

    LightBuffer[LightBuffer_WritePos].lux = lux;
    LightBuffer_WritePos = (LightBuffer_WritePos + 1) % LIGHT_BUFFER_SIZE;
    if (LightBuffer_Count < LIGHT_BUFFER_SIZE) {
        LightBuffer_Count++;
    }
    LightBuffer_DataAvailable = 1;
    __enable_irq();
    return 1;
}

/**
 * @brief Pobiera najnowszy pomiar z bufora
 * @retval Wskaźnik do najnowszego pomiaru lub NULL jeśli brak danych
 */
measurement_entry_t* LightBuffer_GetLatest(void) {
    if (!LightBuffer_DataAvailable) {
        return NULL;
    }
    uint32_t latest_index = (LightBuffer_WritePos == 0)
            ? (LIGHT_BUFFER_SIZE - 1)
            : (LightBuffer_WritePos - 1);
    return &LightBuffer[latest_index];
}

/**
 * @brief Pobiera liczbę pomiarów w buforze
 * @retval Liczba pomiarów
 */
uint32_t LightBuffer_GetCount(void) {
    return LightBuffer_Count;
}

/**
 * @brief Pobiera pomiar z bufora według offsetu (od najnowszego)
 * @param offset: Offset od najnowszego pomiaru (0 = najnowszy)
 * @retval Wskaźnik do pomiaru lub NULL jeśli offset poza zakresem
 */
measurement_entry_t* LightBuffer_GetByOffset(uint16_t offset) {
    if (!LightBuffer_DataAvailable) {
        return NULL;
    }
    uint32_t count = LightBuffer_Count;
    if (count == 0 || offset >= count) {
        return NULL;
    }
    uint32_t actual_index = (LightBuffer_WritePos - 1 - offset + LIGHT_BUFFER_SIZE)
            % LIGHT_BUFFER_SIZE;
    return &LightBuffer[actual_index];
}

/**
 * @brief Pobiera pomiar według indeksu (od najstarszego)
 * @param index: Indeks od najstarszego pomiaru (0 = najstarszy)
 * @retval Wskaźnik do pomiaru lub NULL jeśli index poza zakresem
```

```
 */
measurement_entry_t* LightBuffer_GetByIndexOldest(uint16_t index) {
    uint32_t count = LightBuffer_Count;
    if (count == 0 || index >= count) {
        return NULL;
    }
    uint32_t offset_from_latest = (count - 1U) - index;
    return LightBuffer_GetByOffset((uint16_t)offset_from_latest);
}

```

7. Obsługa błędów czujnika

Funkcje i mechanizmy tej sekcji odpowiadają za wykrywanie, sygnalizowanie i automatyczną obsługę błędów związanych z komunikacją I2C oraz stanem czujnika BH1750. Pozwalają na automatyczne próby ponownej inicjalizacji i minimalizują ryzyko zawieszenia się systemu w przypadku problemów sprzętowych.

```
// Flaga błędu I2C (ustawiana w callbacku błędu)
static __IO uint8_t I2C_Error = 0;

/**
 * @brief Callback błędu I2C
 * Wywoływany automatycznie przez HAL w przypadku błędu transmisji I2C.
 */
void HAL_I2C_ErrorCallback(I2C_HandleTypeDef *hi2c) {
    if (hi2c == &hi2c1) {
        I2C_Error = 1; // Ustaw flagę błędu do obsługi w automacie stanów
    }
}

/**
 * @brief Fragment automatu stanów obsługujący błąd czujnika/I2C
 * (fragment funkcji BH1750_Init_Process)
 */
case BH1750_STATE_ERROR: {
    // Spróbuj ponownie po 2 sekundach
    if ((HAL_GetTick() - bh1750_state_tick) >= 2000) {
        // Reset I2C peripheral jeśli jest w błędnym stanie
        if (hi2c1.State != HAL_I2C_STATE_READY) {
            HAL_I2C_DeInit(&hi2c1);
            HAL_I2C_Init(&hi2c1);
        }
        bh1750_state = BH1750_STATE_IDLE;
        config_step = 0;
    }
    break;
}
```