Politechnika Warszawska

WYDZIAŁ ELEKTRYCZNY

Instytut Elektrotechniki Teoretycznej i Systemów Informacyjno-Pomiarowych

Praca dyplomowa magisterska

na kierunku Elektrotechnika

w specjalności Systemy Wbudowane

Prywatna sieć czujnikowa wykorzystująca standard LoRa

inż. Mikołaj Rosiński

numer albumu 290988

promotor

dr inż. Łukasz Makowski

WARSZAWA 2023

Prywatna sieć czujnikowa wykorzystująca standard LoRa

Streszczenie

Streszczenie po polsku

Słowa kluczowe:

Private sensor network using the LoRa standard

Abstract

Abstract in English

Keywords:

Spis treści

Rozdział 1

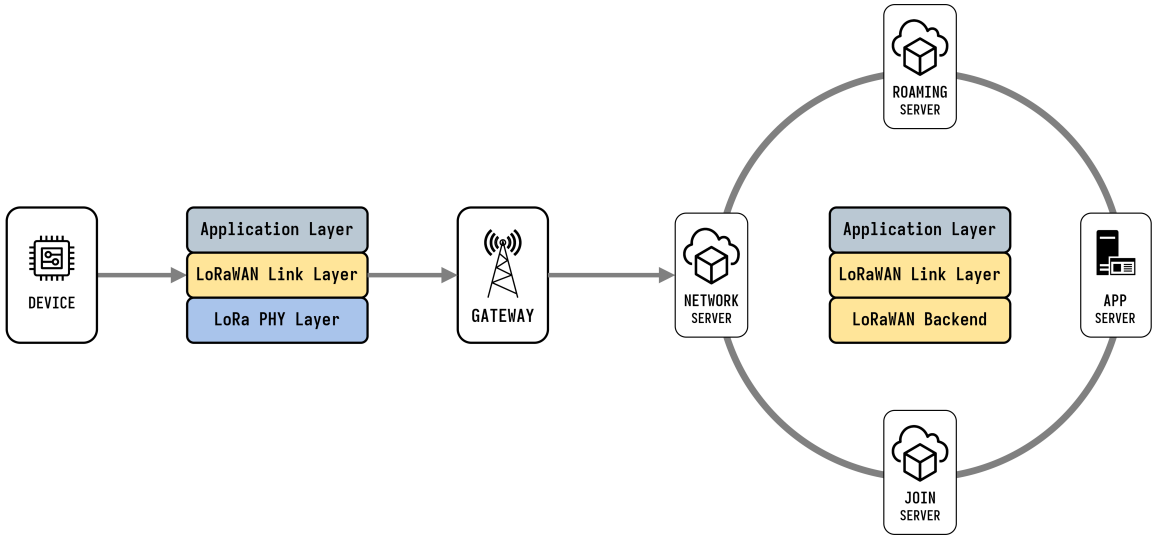
Wstęp

Intro

Rozdział 2

Sieci w standardzie LoRa

2.1 LoRaWAN



Rysunek 1. Schemat architektury sieci LoRaWAN

Rozdział 3

Przygotowanie środowiska programistycznego

Oprogramowanie wszystkich elementów zostało napisane z wykorzystaniem PlatformIO. Narzędzie pozwala na budowanie pod systemy wbudowane na wiele platform [3], w tym wykorzystane do zbudowania sieci STMicroelectronics STM32 Nucleo. Do kompilacji kodu źródłowego możliwe jest użycie wtyczki do edytora Visual Studio Code „PlatformIO IDE" lub samodzielnego narzędzia CLI (ang. *Command Line Interface*) „PlatformIO Core".

Rozwiązanie to zostało wybrane jako główne narzędzie do kompilacji oraz wgrywania kodu źródłowego, z uwagi na to, że działa na wielu plaformach. Dzięki temu nie jest wymagane instalowanie oraz ustawianie osobnych, dedykowanych środowisk dla każdej z wykorzystywanych platform. Jedynym wymogiem, aby móc zacząć pracę jest zainicjowanie projektu oraz ustawienie podstawowej konfiguracji. Zadanie to jest bardzo proste, ponieważ dokumentacja narzędzia jest rozbudowana i bardzo szczegółowa.

3.1 Rozpoczęcie projektu z PlatformIO Core

Całość sieci składa się z dwóch oddzielnych projektów – pierwszy z nich to projekt uniwersalny dla modułów MASTER oraz SLAVE sieci LoRa, drugi natomiast wykorzystywany jest do mikrokontrolera Adafruit Feather M0. Aby rozpocząć nowy projekt, należy wykorzystać komendę, gdzie argumentem jest docelowy mikrokontroler:

```
pio project init --board <board>
```

W przypadku projektu dla sieci LoRa wykorzystane zostały płytki Nucleo L152RE, stąd argumentem było `nucleo_l152`, natomiast dla projektu serwera sieci lokalnej – `adafruit_feather_m0`. Użycie komendy rozpoczyna proces tworzenia nowego projektu. Na podstawie podanego argumentu tworzony jest plik konfiguracyjny. Zdefiniowane zostają platforma projektu oraz wykorzystywany framework. W przypadku obu projektów wybrany został ten wykorzystywany przez Arduino z uwagi

na dużą dostępność bibliotek, które działają bez potrzeby modyfikowania ich kodu źródłowego. Dodatkowo zdefiniowana została tutaj prędkość transmisji portu szeregowego.

W projekcie dla modułów sieci wykonana została modyfikacja pliku konfiguracyjnego – elementy wygenerowane przez narzędzie CLI PlatformIO przeniesione zostały do osobnej sekcji `[base_config]`, natomiast konfiguracje dla poszczególnych modułów znajdują się w dedykowanych „środowiskach". Wprowadzone zmiany zostały dokładniej opisane w sekcjach o implementacji oprogramowania na poszczególne moduły (4.2, 4.3).

Poza plikiem konfiguracyjnym, narzędzie generuje też podstawową strukturę plików całego. Powstaje folder `src`, który dedykowany jest dla plików źródłowych, `include` dla plików nagłówkowych, `lib` dla bibliotek lokalnych oraz `tests` do testów jednostkowych, jeżeli planowane jest użycie ich.

3.2 Praca z PlatformIO

Po stworzeniu projektu możliwe jest przystąpienie do pisania kodu źródłowego na wybraną platformę. PlatformIO udostępnia możliwość kompilowania kodu oraz wgrywania go na docelowe urządzenie poprzez jedną jedną komendę lub jeden przycisk w edytorze tekstu. Jest to bardzo dobre rozwiązanie, ponieważ dzięki temu możliwe jest skupienie się na rozwoju kodu źródłowego, zamiast czekania aż projekt będzie możliwy do uruchomienia i sprawdzenia.

3.2.1 Uruchamianie projektu

Uruchomienie projektu jest w przypadku PlatformIO rozumiane poprzez wykonanie kompilacji (`build`), wgranie skompilowanego kodu na urządzenie docelowe (`upload`) lub wykonanie zdefiniowanego zestawu testów jednostkowych (`test`). Aby uruchomić projekt należy wykorzystać komendę:

```
pio run [OPTIONS]
```

Argumentami dodatkowymi mogą być:

- `--environment`: element konfiguracji projektu, który określa zależności w kwestiach kompilacji (np. flagi budowania projektu), programowania (wgrywania kodu) docelowych urządzeń, testów jednostkowych lub wykorzystanych bibliotek,
- `--target`: cel uruchomienia (np. kompilacja albo kombinacja kilku celów jednocześnie),
- `--upload-port`: port, do którego podłączone jest urządzenie i na które ma zosatać wgrany kod. Szczególnie użyteczne w przypadku, gdy pracuje się na wielu urządzeniach jednocześnie,
- `--monitor-port`: port, na którym po zakończeniu procesu ma zostać otwarty monitor portu szeregowego.

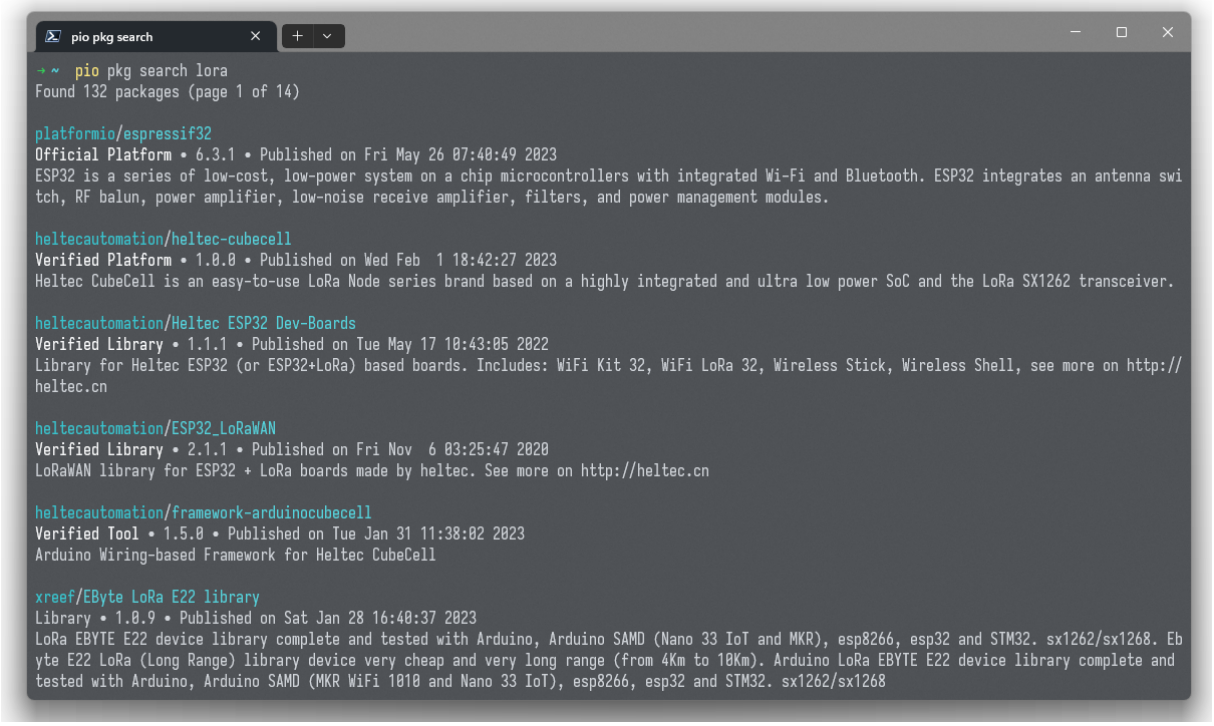
W przypadku opcji związanych z portem, jeżeli nie zostaną sprecyzowane (podane jako argument do komendy), PlatformIO będzie próbował wykryć je automatycznie. Dostępne jest jeszcze kilka innych opcji, jednkże są one znacznie rzadziej wykorzystywane, ponieważ ich domyślne opcje są tymi, które są najczęściej ustawiane.

3.2.2 Zarządzanie bibliotekami

PlatformIO posiada wbudowany moduł dedykowany do zarządzania bibliotekami oraz innymi zasobami dołączanymi do projektu. Dzięki wykorzystaniu odpowiedniej podkomendy z zestawu:

```
pio pkg [COMMAND]
```

możliwe jest przeszukiwanie, instalowanie z, aktualizacja lub publikowanie do rejestru dostępnych bibliotek. Podczas wyszukiwania możliwe jest też zastosowanie filtrów, które w znacznym stopniu zmniejszają ilość wyników i przybliżają do znalezienia tego pasującego. Wykorzysując tę operację zainstalowane zostały potrzebne do projektów biblioteki (wbudowane dla frameworku Arduino, tak jak „Wire" czy te, które opublikowane zostały na platformie GitHub i dodane do rejestru PlatformIO). Na rys. 2 przedstawiony zostały przykładowy wynik wyszukiwania dostępnych bibliotek związanych z hasłem „LoRa". Każdy wynik zawiera informację: nazwę, typ paczki, biblioteki, która została znaleziona, najnowszą wersję, datę publikacji oraz krótki opis tego czym dana paczka, biblioteka są. Komenda pokazuje także informacje o tym ile wyników zostało znalezionych.



Rysunek 2. Wyniki wyszukiwania bibliotek powiązanych z hasłem „LoRa"

Rozdział 4

Implementacja oprogramowania

Całość oprogramowania wykorzystuje język programowania C++. Projektowana oraz implementowana sieć składa się z dwóch typów modułów, stąd też pojawiła się potrzeba zainicjowania dwóch osobnych projektów – jednego pod elementy sieci LoRa oraz drugiego, dedykowanego dla modułu serwera sieciowego (ang. *webserver*), z uwagi na zupełnie inną platformę sprzętową. Firmware napisany został z wykorzystaniem kilku różnych podejść:

- modułowego: każdy plik źródłowy odpowiada za zbiór funkcji wykonujących określone zadania (np. praca z biblioteką do modułów LoRa zaimplementowana jest w pliku `lora.cpp`),
- obiektowego: większość elementów kodu źródłowego jest reprezentowana w postaci osobnego obiektu. Każdy z nich posiada swoje funkcje oraz pełni określone zadania (np. obiekt „`bme`" ma za zadanie umożliwić współpracę z sensorami dostępnymi na płytce czujników BME280, która podłączona jest do każdego modułu SLAVE).

Ponadto, wykorzystane zostały elementy języka C++, które dostępne są w nowszych wersjach – funkcje szablonowe (ang. *template functions*) lub pętle typu `for-range`. Są to elementy, które znacznie ułatwiły implementację kodu oraz pozwoliły na minimalizację powtarzalności pewnych elementów.

Z uwagi na zastosowanie podejścia modułowego, całość oprogramowania składa się z wielu mniejszych elementów, podzielonych na odpowiadające im pliki. Aby mieć pewność, że implementowane funkcje nie będą posiadały żadnych kolizji w swoich nazwach, zastosowane zostały przestrzenie nazw (ang. *namespaces*). Co więcej, ponieważ kod źródłowy jest dostępny w domenie publicznej (repozytorium na platformie GitHub z licencją *MIT* [4]), podjęta została decyzja o dodaniu opisów działania do wszystkich elementów. Wykorzystany został do tego *Doxygen* – narzędzie do generowania dokumentacji (np. formie strony internetowej lub dokumentu w LaTeX) na podstawie specjalnych znaczników w komentarzach [2].

4.1 Framework oraz biblioteki

Bazą do oprogramowania na wszystkich modułach jest framework Arduino oraz jego modyfikacja pod platformę STM32 – stm32duino, która pozwala na wykorzystanie pełnej funkcjonalności rdzenia

Arduino [5]. Pomimo tego, że biblioteki HAL (ang. *Hardware Abstraction Layer*) oraz framework STM32 są narzędziami dedykowanymi, w przypadku tego projektu nie można było ich zastosować. Oryginalna biblioteka do obsługi modułów rozszerzeń LoRa została wycofana z użytku na rzecz nowszej implementacji, pod nowszą wersję płytek Nucleo z wbudowanym hardware.

4.1.1 Wykorzystane biblioteki

Do implementacji oprogramowania na wszystkie moduły wykorzystanych zostało kilka bibliotek, które pozwalały na dodanie pełnego zakresu funkcjonalności do każdego z projektów.

W przypadku bibliotek zewnętrznych (niebędących częścią rdzenia Arduino) były to:

- STM32duino I-NUCLEO-LRWAN1: biblioteka do uruchomienia oraz pracy z modułem rozszerzeń LoRa. Pozwala ona na pracę w dwóch trybach: LoRaRadio – implementacja wykorzystująca tylko standard dolnej warstwy sprzętowej LoRa oraz LoRaWAN – dodająca możliwość podłączenia modułów do istniejącej sieci LoRa oraz wysyłanie i odbieranie z niej wiadomości,
- Adafruit BME280 Library: biblioteka dedykowana do modułów BME280, pozwalająca na zbieranie danych z sensorów, wykorzystując do tego magistralę SPI albo I2C (w zależności od posiadanego modułu rozszerzeń),
- Adafruit BusIO: uniwersalna biblioteka dodająca pewien poziom abstrakcji do komunikacji po magistralach I2C oraz SPI,
- WiFi101: biblioteka, która daje możliwość wykorzystania modułu WiFi obecnego na płytce Adafruit Feather M0 (wykorzystanej do uruchomienia serwera w sieci lokalnej).

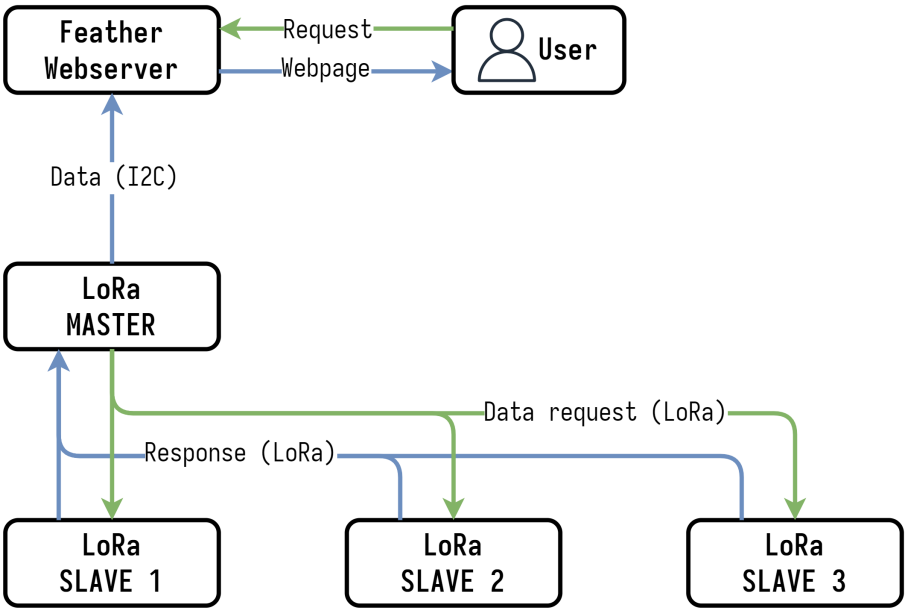
Ponadto, wykorzystane zostały biblioteki I2C oraz SPI, dostępne w rdzeniu Arduino. Potrzebne były one do uzyskania komunikacji pomiędzy mikrokontrolerem Adafruit Feather M0 a modułem WiFi, sensorami BM280 podłączonymi do modułów SLAVE oraz do stworzenia połączenia pomiędzy modułem MASTER a płytką z serwerem sieci lokalnej.

4.1.2 Ograniczenia związane z wykorzystaniem Arduino oraz STM32duino

STM32duino, pomimo tego, że ułatwił, bądź w ogóle pozwolił na pracowanie z wykorzystywanymi modułami, nie jest platformą idealną, pozbawioną ograniczeń. Jedynym z nich, które w dość znacznym stopniu utrudniło implementację oprogramowania dla modułów sieci, był brak przerwań programowych oraz ograniczone możliwości zastosowania przerwań sprzętowych. Stąd też pojawił się wymóg zastosowania pewnych obejść, jednocześnie tracąc na wydajności implementowanego rozwiązania. Ponadto, występowały też problemy związane z działaniem magistrali I2C, tutaj w przypadku modułów Feather oraz standardowego Arduino – niemożliwe było wykorzystanie wyświetlacza OLED pracującego na magistrali I2C oraz zarejestrowania samego mikrokontrolera jako części, z którą można komunikować się po tej magistrali.

4.2 Implementacja oprogramowania elementów sieci

Zaprojektowana sieć składała się w sumie z pięciu modułów – 4 z nich stanowiły elementy sieci LoRa, natomiast ostatni był wykorzystywany jako serwer w sieci lokalnej. W projekcie nie została wykorzystana pełna funkcjonalność LoRaWAN oraz typowa dla niej architektura (przedstawiona w sekcji 2.1, rys. 1), ponieważ implementacja takiego rozwiązania jest bardzo kosztowna i wymaga znacznie większej ilości elementów. Aby móc skorzystać ze specyfikacji wymagane jest posiadanie bramy (ang. *gateway*) oraz serwerów odpowiedzialnych za przyłączanie urządzeń, zarządzanie siecią oraz serwera aplikacyjnego. Z uwagi na to zastosowana została dużo prostsza i mniej wymagająca metoda budowania sieci, opierająca się na wykorzystaniu modułów w formie nadajników radiowych, pracujących w standardzie LoRa. Schemat ideowy budowanej sieci przedstawiony został na rys. 3.



Rysunek 3. Schemat zbudowanej sieci, z oznaczonymi elementami komunikacji

Oprogramowanie dla modułów pracujących w sieci LoRa zostało zaimplementowane w formie uniwersalnej – jeden projekt zawiera elementy dla modułu MASTER oraz modułów SLAVE. Plik konfiguracyjny projektu zawiera flagę, która definiuje, na jaki typ modułu kod zostanie skompilowany. Co więcej, w przypadku modułów SLAVE dodana została też flaga informująca o tym, jakie ID przypisane zostaje danej płytce. Rozwiązanie to odgrywa znaczącą rolę w tym, jak wiadomości są przesyłane w sieci. Fragment pliku konfiguracyjnego, który odpowiedzialny jest za definiowanie tych elementów przedstawiony został na listingu 1.

Wykorzystanie frameworku Arduino wymagało zastosowania pewnych schematów podczas implementacji. Dlatego też całość kodu podzielona jest na dwie sekcje `setup()` oraz `loop()`, wykonywane odpowiednio raz, podczas startu modułu oraz w nieskończonej pętli, dopóki płytka ma zasilanie. Na

```
[env:SLAVE1]
extends = base_config
build_flags =
  -DBOARD_TYPE=lora::SLAVE
  -DBOARD_ID=0x01
```

Listing 1. Fragment pliku konfiguracyjnego (tutaj dla SLAVE1) odpowiedzialny za definicję typu oraz ID modułu

rys. 4 przedstawiony został schemat blokowy zaimplementowanego oprogramowania – części zawartej w sekcji `setup()`.

Oba typy oprogramowania zaczynają od ustawienia portu szeregowego na 115200 baud (szybkość transmisji), następnie inicjowane jest rozszerzenie LoRa. Logowana jest informacja o typie płytki, a następnie kod oczekuje na informacje o starcie modułu rozszerzenia. W przypadku błędu oraz poprawnego startu na port szeregowy wystawiana jest odpowiednia informacja.

Następnie, w zależności od typu płytki, wykonywane jest kilka operacji. W przypadku modułów SLAVE są to:

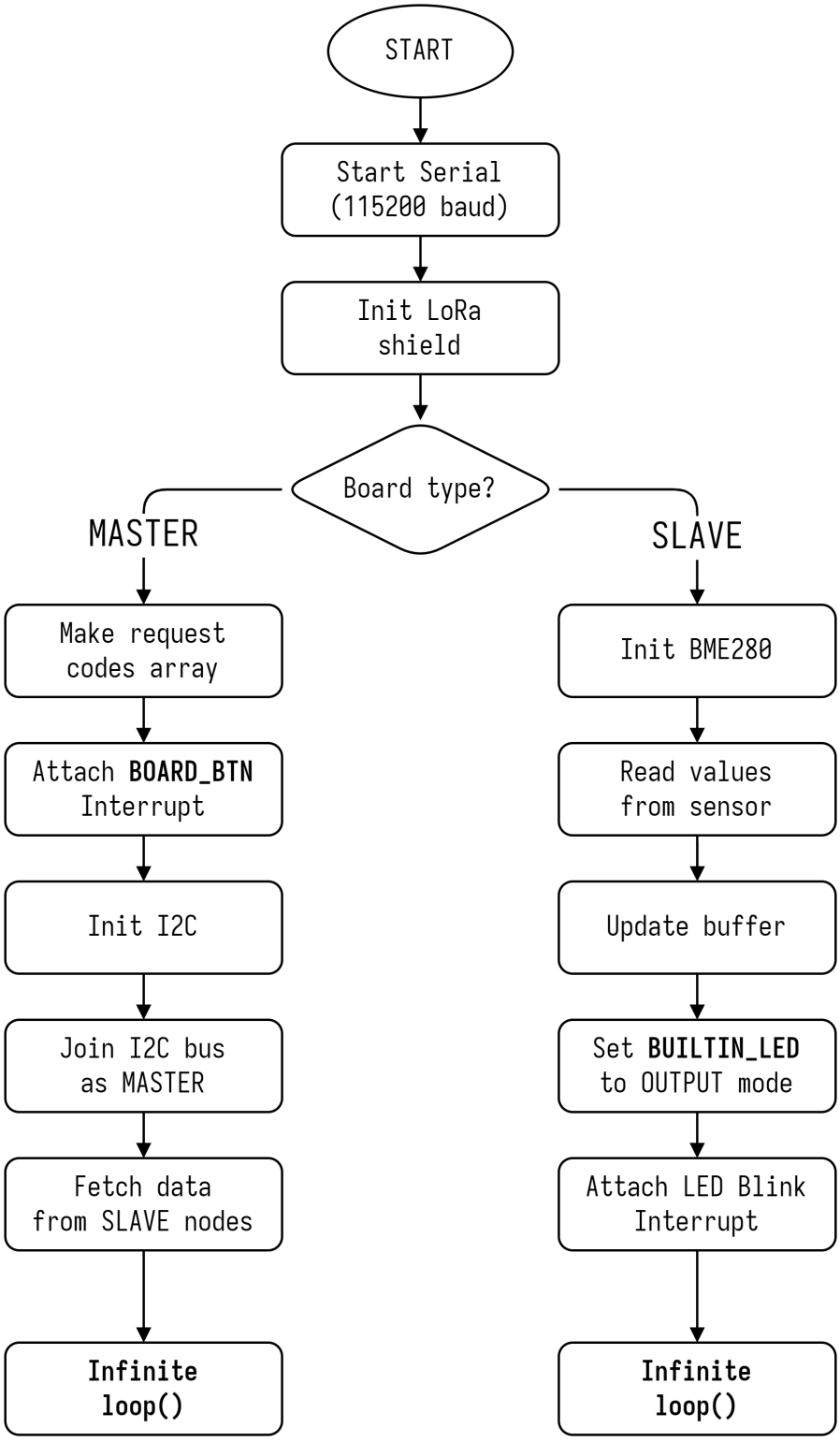
1. przygotowanie sensora BME280 oraz pobranie z niego danych,
2. aktualizacja zawartości bufora (wykorzystywanego do przechowywania odczytanych wartości),
3. przygotowanie diody LED, która informuje o trwającej komunikacji w sieci,
4. przygotowanie przerwania, wykorzystywanego do obsługi nowych zapytań.

Natomiast dla modułów MASTER wykonywany jest inny zestaw operacji, z uwagi na to, że taki moduł pełni zupełnie inną funkcję w sieci:

1. przygotowanie tablicy z „kodami" zapytań (jedno bajtowe wartości do określenia czego żąda MASTER),
2. inicjacja magistrali I2C i podłączenie modułu jako MASTER,
3. wykonanie podprogramu wysyłającego zapytania oraz odbierającego odpowiedzi od SLAVE-ów, tak aby tuż po starcie można było odczytać dane z sieci.

Ostatnim krokiem w obu przypadkach jest przejście do nieskończonej pętli i wykonywanie instrukcji w niej zawartych, wykorzystując do tego określony okres zegara.

Ponadto, oprogramowanie posiada zestaw definicji oraz funkcji wykorzystywanych do debugowania, które ułatwiały implementację oprogramowania – `globals.h` oraz `debug.h`. Najważniejszymi elementami pliku globalnych definicji są funkcje preprocesora – zwracających tylko ID modułu lub ID danej na podstawie kodu zapytania oraz struktury szablonowe (ang. *template structures*), które zawierają informację o tym jaki kształt powinny mieć dane zbierane z sensorów oraz przekazywane



Rysunek 4. Schemat blokowy części `setup()` oprogramowania modułów sieci LoRa, z podziałem na typ płytki

przez sieć. Definicję przedstawiono na listingu 2, natomiast dla funkcji wysyłającej sformatowane wiadomości przez port szeregowy na listingu 3. Implementacja oparta została o funkcję z rdzenia Arduino – `Serial.println()`.

```
#define DATAID_MASK(req) (req & 0xf0)
#define BOARDID_MASK(req) (req & 0x0f)
#define ARRAYSIZE(x) (sizeof(x) / sizeof(x[0]))

/**
 * @brief Template struct for single value sensor data.
 * @tparam T Field variable type
 */
template <typename T>
struct SensorValues
{
    T temperature;
    T pressure;
    T humidity;
};

/**
 * @brief Template struct for sensor data that needs an array.
 * @tparam T Field variable type
 * @tparam N Array size
 */
template <typename T, size_t size>
struct SensorData
{
    T temperature[size];
    T pressure[size];
    T humidity[size];
};
```

Listing 2. Definicje funkcji dla preprocesora oraz struktury szablonowe (z polami o jednej wartości oraz z tablicami)

```
/**
 * @brief Prints out a debug message line over Serial
 *
 * @param type: Debug message type @ref MsgType_t
 * @param msg: Message string
 */
void println(MsgType_t type, String msg);
```

Listing 3. Funkcji wykorzystywana do wysyłania sformatowanych wiadomości przez port szeregowy

4.2.1 Oprogramowanie modułu MASTER

Po wykonaniu instrukcji, które opisane zostały w poprzedniej sekcji, moduł MASTER przechodzi do pracy w nieskończonej pętli – `loop()`. Wszystko oparte jest na zegarze o zdefiniowanym okresie – wybrana została wartość 1 minuty (60000 milisekund). Implementacja oparta została o zegar nieblokujący (ang. *non-blocking timer*) z wykorzystaniem funkcji `millis()` – funkcji zwracającej ilość milisekund od momentu startu programu. Okres został zdefiniowany w definicjach preprocesora, w celu uniknięcia tzw. magicznych liczb (ang. *magic numbers*).

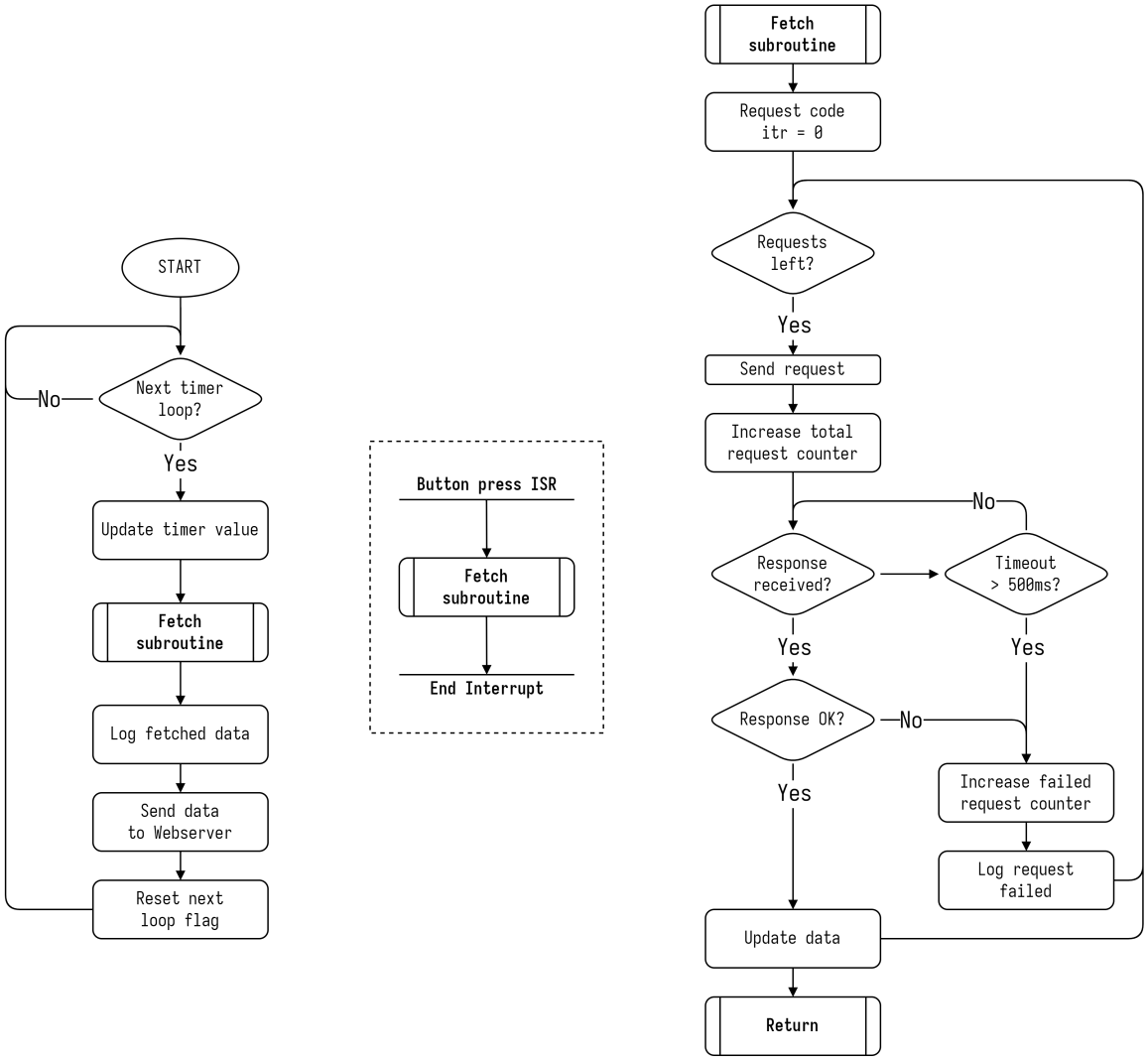
W momencie, gdy mija wymagany czas, program przechodzi do wykonania podprogramu odpowiadającego za wysyłanie zapytań oraz zbieranie odpowiedzi z sieci. Na rys. 5 przedstawiony został diagram blokowy instrukcji wykonywanych przez moduł MASTER w nieskończonej pętli oraz tego, co wykonywane jest w podprogramie komunikacji. Natomiast pełna implementacja obu tych elementów przedstawiona została na listingach 4 oraz 5.

```
case lora::MASTER:
    if (next)
    {
        timer = millis();
        fetchSubroutineHandler();
        INVERT(next);
    }
    break;

if ((millis() - timer) >= PERIOD_MS)
{
    INVERT(next);
}
```

Listing 4. Implementacja nieskończonej pętli dla modułu MASTER

Pierwszym elementem podprogramu jest wysłanie nowego zapytania do sieci – zaimplementowana została do tego funkcja `sendRequest()` zawarta w przestrzeni nazw `lora`. Jej implementacja przedstawiona została na listingu 6. Pobierany jest kod, który ma zostać wysłany do sieci, następnie, wykorzystując funkcję `debug::println()` logowana jest przesyłana wartość. Korzystając z funkcji,



Rysunek 5. Schemat blokowy nieskończonej pętli oraz podprogramu zbierania danych zaimplementowanych dla modułu MASTER

```
void fetchSubroutineHandler(void)
{
    for (u8 code : requestCode)
    {
        fetchData(code);
    }

    logReceivedData(&receivedData);
    webserverTransmit(&receivedData);
}

void fetchData(u8 requestCode)
{
    lora::sendRequest(requestCode);
    totalRequests[(BOARDID_MASK(requestCode)) - 1] += 1;

    // If MASTER waits 500ms for response, treat fetch as failed
    u32 timeout = millis();
    while (!(loraRadio.read(receivedMsg) > 0))
    {
        if ((millis() - timeout) >= TIMEOUT_MS)
        {
            timeout = millis();
            debug::println(debug::ERR, "Request 0x" + String(requestCode,
                HEX) +
                                                " failed: TIME OUT.");
            failedRequests[(BOARDID_MASK(requestCode)) - 1] += 1;
            return;
        }
    }

    // No timeout, check response code matches request code
    if (receivedMsg[0] != requestCode)
    {
        debug::println(debug::ERR, "Request 0x" + String(requestCode, HEX)
            +
                                            " failed: BAD RESPONSE");
        failedRequests[(BOARDID_MASK(requestCode)) - 1] += 1;
        return;
    }

    lora::readResponse(&receivedData, receivedMsg);
    delay(100); // 100ms blocking delay between requests
}
```

Listing 5. Funkcja podprogramu odpowiedzialnego za zbieranie danych w sieci

która dostępna jest w bibliotece do obsługi modułu rozszerzeń LoRa, wysyłana jest wiadomość do sieci.

```
1   void sendRequest(u8 message)
2   {
3       u8 msg[] = {message};
4       debug::println(debug::INFO, "Sending new requst with value 0x" +
            String(message, HEX));
5
6       loraRadio.write(msg, sizeof(msg));
7   }
```

Listing 6. Implementacja funkcji `lora::sendRequest()`

Następnie moduł MASTER oczekuje na odpowiedź od modułu SLAVE, który powinien wysłać odpowiedź. Jeżeli odpowiedź zostana otrzymana w ciągu 500ms, następuje przejście do sprawdzenia, czy pierwsze pole odpowiedzi – identyfikator – jest poprawne. Identyfikator zawiera informację o ID odpowiadającego SLAVE-a oraz ID danej, której wartość jest przesyłana. W przeciwnym razie, na port szeregowy przesyłana jest stosowna informacja, a licznik zapytań z błędem odpowiedzi jest zwiększany. Ostatecznie, jeżeli nie wystąpił żaden z błędów, wykorzystując funkcję `lora::readResponse()`, odczytana zostaje wartość przesłana w odpowiedzi. Implementacja funkcji odczytującej przedstawiona została na listingu 7.

```
1   void readResponse(ReceivedData *data, u8 message[])
2   {
3       u8 boardId = BOARDID_MASK(message[0]) - 1;
4       // Merge each 2x 8-bit fields into 1x 16-bit one, fix magnitudes
5       switch (DATAID_MASK(message[0]))
6       {
7       case TEMPERATURE:
8           data->temperature[boardId] = (f32)((message[1] << 8) + message
                [2]) / 100;
9           break;
10
11      case PRESSURE:
12          data->pressure[boardId] = (f32)((message[1] << 8) + message[2])
                ;
13          break;
14
15      case HUMIDITY:
16          data->humidity[boardId] = (f32)((message[1] << 8) + message[2])
                / 100;
17          break;
18      }
19  }
```

Listing 7. Implementacja funkcji odczytującej wartość odpowiedzi modułu SLAVE

W funkcji sprawdzane są ID modułu, który odpowiedź wysłał oraz ID danej. Na podstawie tej wartości, aktualizowana jest odpowiednia indeks w tablicy, która odpowiada polu struktury do przechowywania danych odbieranych z sieci. Struktura ta przekazywana jest jako referencja do miejsca w pamięci poprzez wskaźnik do jej adresu.

Ostatnimi elementami każdej iteracji pętli jest przesłanie zebranych danych przez port szeregowy oraz transmisja danych do modułu pełniącego funkcję serwera sieciowego. Funkcja logowania danych przez port szeregowy została dodana, po to, aby było możliwe debugowanie działania oprogramowania oraz naprawa ewentualnie występujących błędów. Do implementacji wykorzystana została wykorzystana funkcja szablonowa, która pozwoliła na wykorzystanie tego samego fragmentu kodu do przesyłania wartości z tablic o różnym typie zmiennej (`float` – zmiennoprzecinkowa – dla wartości pochodzących z sieci oraz `int` – liczby całkowite – dla wartości związanych ze statystykami zapytań). Na funkcje wykorzystywane do transmisji danych przez magistralę I2C do modułu serwera sieciowego składa się kod zaimplementowany, korzystając z tego samego schematu. Przesyłanie wartości z pojedynczego pola struktury wykorzystuje także funkcję szablonową, która wywoływana jest kilkukrotnie wewnątrz `webserverTransmit` w celu przesłania wszystkich wymaganych danych. Kod funkcji szablonowych przedstawiony został na listingach 8 oraz 9, natomiast implementacja pełnych funkcji do przesyłania danych na listingach 10 oraz 11. Dodatkowo zaimplementowana została także funkcja pomocnicza do wyznacznia wartości procentowej zapytań, które zakończyły się błędem. Opiera się ona o wykonanie dzielenia wartości z licznika zapytań z błędem (`failedRequests`) przez wartość licznika całkowitej ilości zapytań wysłanych do każdego z modułów SLAVE (`totalRequests`). Kod tej funkcji przedstawiony został na listingu 12.

```
template <typename T, size_t size>
extern void logValues(const T (&array)[size])
{
    for (T item : array)
    {
        Serial.print(item);
        Serial.print("\t");
    }
    Serial.println();
}
```

Listing 8. Implementacja funkcji szablonowej `logValues()`

```
template <typename T, size_t size>
void transmitPacket(const T (&array)[size], u8 typeKey, f32 modifier)
{
    char packet[50];
    for (T item : array)
    {
        sprintf(packet, "%i:%i&", typeKey, (u16)(item * modifier));
        Wire.write(packet);
    }
}
```

Listing 9. Implementacja funkcji szablonowej `transmitPacket()`

```
void logReceivedData(lora::ReceivedData *data)
{
    Serial.println();
    debug::println(debug::INFO, "Fetched data:");

    Serial.print("Temperature:\t");
    logValues(data->temperature);
    Serial.print("Pressure:\t");
    logValues(data->pressure);
    Serial.print("Humidity:\t");
    logValues(data->humidity);
    Serial.println();

    debug::println(debug::INFO, "Requests statistics:");

    Serial.print("Total:\t\t");
    logValues(totalRequests);
    Serial.print("Failed:\t\t");
    logValues(failedRequests);
    getFailedPercent(totalRequests, failedRequests, failedPercent);
    Serial.print("Failed%:\t");
    logValues(failedPercent);

    Serial.println();
}
```

Listing 10. Funkcja wykorzystywana do logowania wartości przez port szeregowy

```
void webserverTransmit(lora::ReceivedData *data)
{
    debug::println(debug::INFO, "Sending to webserver");
    Wire.beginTransmission(I2C_ADDR);

    transmitPacket(data->temperature, TEMPERATURE, 100.0f);
    Serial.println("Temperature");
    transmitPacket(data->pressure, PRESSURE);
    Serial.println("Pressure");
    transmitPacket(data->humidity, HUMIDITY, 100.0f);
    Serial.println("Humidity");
    getFailedPercent(totalRequests, failedRequests, failedPercent);
    transmitPacket(failedPercent, FAILPERCENT, 100.0f);
    Serial.println("Failed%");

    Wire.endTransmission();
}
```

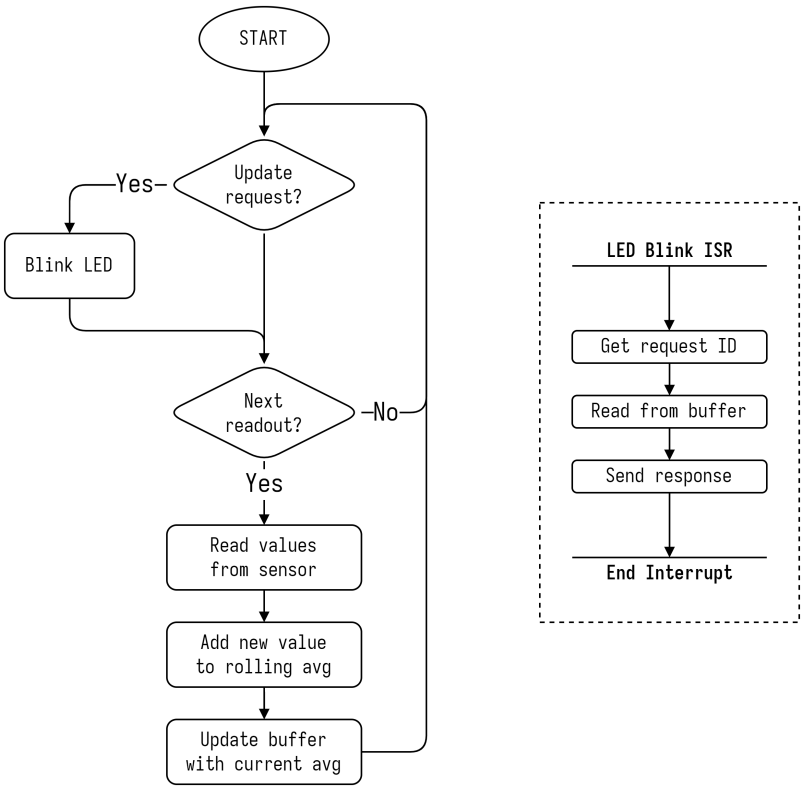
Listing 11. Funkcja do transmisji danych do modułu serwera przez magistralę I2C

```
void getFailedPercent(u8 (&totalReq)[3], u8 (&failReq)[3],
                      f32 (&failPercent)[3])
{
    for (u8 i = 0; i < sizeof(totalReq); ++i)
    {
        failedPercent[i] = ((f32)failReq[i] / (f32)totalReq[i]) * 100.0f;
    }
}
```

Listing 12. Implementacja funkcji pomocniczej do wyznaczania wartości procentowej zapytań z błędem

4.2.2 Oprogramowanie modułów SLAVE

W przypadku modułów SLAVE działanie kodu w nieskończonej pętli zostało zaimplementowane inaczej. Poza wykonywaniem zadań bazując na zegarze, w tym przypadku z okresem 5 sekund, zaimplementowane zostały także przerwania. Wykorzystywane są one do obsługi przychodzących nowych zapytań od modułu MASTER. Tak jak zostało to opisane w sekcji 4.1.2, we frameworku Arduino nie ma możliwości wykorzystania przerwań programowych, dlatego też zostało zastosowane obejście bazujące na przerwaniu sprzętowym. Na rys. 6 przedstawiony został diagram nieskończonej pętli zaimplementowanej dla modułów SLAVE.



Rysunek 6. Diagram blokowy pętli `loop()` zaimplementowanej dla modułów SLAVE

Zadania wykonywane przez moduły, wykorzystując do tego zegar (zaimplementowany w sposób identyczny do modułu MASTER – na bazie funkcji `millis()`), to głównie zbieranie danych z sensorów płytki BME280 oraz akutualizacja bufora z danymi. Implementacja tego fragmentu kodu została przedstawiona na listingu 13.

Odczytywanie danych z sensora odbywa się wykorzystując funkcję `sensor::readRaw()`, która przedstawiona jest na listingu 14. Jedynym jej zadaniem jest odczytanie wszystkich potrzebnych danych oraz dodanie tych wartości do tablicy dla średniej kroczącej.

Obliczanie średniej kroczącej rozwiązane zostało poprzez zainicjowanie zaimplementowanej do tego klasy – `RollingAvg`. Jej definicja przedstawiona została na listingu 15. W celu uzyskania

```
    case lora::SLAVE:
        if (loraRadio.read(updateRequestMsg))
        {
            // This will trigger an interrupt
            digitalWrite(LED_BUILTIN, 1);
        }

        if (next)
        {
            sensor::readRaw();
            sensor::updateBuffer(&sensorBuffer);
            timer = millis();
            INVERT(next);
        }
        break;

    if ((millis() - timer) >= PERIOD_MS)
    {
        INVERT(next);
    }
```

Listing 13. Implementacja nieskończonej pętli dla modułów SLAVE

```
    void readRaw(void)
    {
        temperatureAvg.addValue(bme.readTemperature());
        pressureAvg.addValue(bme.readPressure());
        humidityAvg.addValue(bme.readHumidity());
    }
```

Listing 14. Implementacja funkcji `sensor::readRaw()` do odczytywania wartości z sensora BME280

optymalnego działania kodu – średnia krocząca jest zadaniem dość intensywnym obliczeniowo – zastosowane zostało przydzielanie pamięci (ang. *memory allocation*) poprzez `malloc()` oraz wypełnianie tablicy samymi zerami przy początkowej deklaracji obiektu, dzięki czemu dodawanie wartości jest znacznie szybsze, ponieważ algorytm nie musi każdorazowo powiększać rozmiaru tablicy. Podczas dodawania wartości do średniej wykonywane jest kilka kroków: sprawdzenie, czy tablica na pewno istnieje, odjęcie od sumy wartości z obecnego indeksu oraz zastępowanie jej nową, zwiększanie indeksu lub zerowanie go, jeżeli wskaźnik dotarł do maksymalnej wartości. Działanie algorytmu przedstawione zostało na rys. 7, natomiast na listingu 16 jego implementacja.

```
class RollingAvg
{
public:
    RollingAvg(u8 nSample);
    ~RollingAvg();

    void clear(void);
    void addValue(f32 value);
    f32 getAverage(void);

private:
    u8 _size;
    u8 _count;
    u8 _idx;
    float _sum;
    float *_array;
};
```

Listing 15. Definicja klasy `RollingAvg`

```
void RollingAvg::addValue(f32 value)
{
  if (_array == NULL)
    return;

  _sum -= _array[_idx];
  _array[_idx] = value;
  _sum += _array[_idx];
  ++_idx;

  if (_idx == _size) // Set _idx to 0 if max reached
    _idx = 0;

  if (_count < _size)
    ++_count;
}
```

Listing 16. Implementacja algorytmu dodającego nową wartość do średniej kroczącej

Rysunek 7. Schemat działania algorytmu dodawania nowej wartości do średniej kroczącej dla przykładowej tablicy 10 elementów, indeksu 2 oraz nowej wartości 38

Bufor na dane został dodany w celu przyspieszenia działania oprogramowania. Dzięki zastosowaniu takiego rozwiązania, w momencie pojawienia się nowego zapytania w sieci, SLAVE, do którego jest ono skierowane, może prawie natychmiast wysłać odpowiedź. Aktualizacja bufora z zebranymi danymi (wartością średniej kroczącej) odbywa się poprzez zmianę ich wartości pól, wykorzystując do tego wskaźniki do obecnych wartości oraz zaimplementowaną funkcję `getAverage()` z klasy `RollingAvg`. Podczas pobierania wartości średniej pierwszym krokiem jest sprawdzenie, czy tablica wartości nie jest pusta, następnie zwracana jest wartość – dzielenie aktualnej sumy przez ilość elementów w tablicy. Implementacja tych funkcji przedstawiona została na listingach 17 oraz 18.

```
    void updateBuffer(BufferData *buffer)
    {
        buffer->temperature = (u16)(temperatureAvg.getAverage() * 100.0f);
        buffer->pressure = (u16)(pressureAvg.getAverage() / 100.0f);
        buffer->humidity = (u16)(humidityAvg.getAverage() * 100.0f);
    }
}
```

Listing 17. Implementacja funkcji aktualizującej zawartość bufora na dane

```
float RollingAvg::getAverage()
{
  if (_count == 0)
    return NAN;

  return (_sum / _count);
}
```

Listing 18. Implementacja funkcji do wyznaczania średniej na podstawie obecnej zawartości tablicy średniej kroczącej

Obejściem limitów stawianych przez framework Arduino było dodanie przerwania sprzętowego. Pin wbudowanej diody mikrokontrolerów (`LED_BUILTIN`, pin PA5) połączony został z dowolnym pinem, który nie pełnił żadnej innej funkcji (tutaj wybrany został pin PB3, zdefiniowany w kodzie jako `SLAVE_INTERRUPT_PIN`). W momencie, gdy moduł rozszerzeń LoRa wykryje nadawaną w sieci dowolną wiadomość, włączona zostaje dioda, co powoduje wejście w przerwanie.

Aktywacja przerwania powoduje wywołanie funkcji `newRequestHandler()`, w której pierwszym elementem jest sprawdzenie, czy zaobserwowane w sieci zapytanie dotyczy danego modułu. Zrealizowane zostało to za pomocą maski, która pozwala na wydobycie tylko ID modułu z całego kodu zapytania. Jeżeli kod jest zgodny, to moduł wysyła odpowiedź, wykorzystując do tego funkcję `lora::sendResponse()`, a następnie wyłączana jest dioda. W przeciwnym przypadku wyłączona zostanie tylko dioda, a moduł wyjdzie z przerwania i powróci do wykonywania instrukcji z nieskończonej pętli. Implementacja funkcji obsługi przerwania przedstawiona została na listingu 19, natomiast na listingu 20 przedstawiona została funkcja zajmująca się wysyłaniem odpowiedzi.

```
void newRequestHandler(void)
{
    if (BOARDID_MASK(updateRequestMsg[0]) != BOARD_ID)
    {
        digitalWrite(LED_BUILTIN, 0);
        return;
    }

    lora::sendResponse(&sensorBuffer, updateRequestMsg[0]);
    digitalWrite(LED_BUILTIN, 0); // Turn off the LED when done, visual
        indicator
}
```

Listing 19. Implementacja funkcji obsługującej przerwania w modułach SLAVE

```
    void sendResponse(sensor::BufferData *buffer, u8 reqMsg)
    {
        /**
         * @brief Payload of the response. First byte is request code echo,
         * then 2 bytes of response, MSB-order.
         */
        u8 message[3];
        u16 bufferValue;

        switch (DATAID_MASK(reqMsg))
        {
        case TEMPERATURE:
            bufferValue = (u16)buffer->temperature;
            break;

        case PRESSURE:
            bufferValue = (u16)buffer->pressure;
            break;

        case HUMIDITY:
            bufferValue = (u16)buffer->humidity;
            break;
        }

        message[0] = reqMsg;
        message[1] = (bufferValue & UPPER_BITMASK) >> 8;
        message[2] = (bufferValue & LOWER_BITMASK);
        debug::println(debug::INFO, "Sending response: 0x" + String(message
            [0], HEX) + "\t" + String(message[1]) + "\t" + String(message
            [2]));

        loraRadio.write(message, sizeof(message));
    }
```

Listing 20. Implementacja funkcji do wysyłania odpowiedzi przez moduły SLAVE

Funkcja `lora::sendResponse()` także bazuje na wykorzystaniu maski, tym razem do sprawdzenia ID danej jakiej wartość powinna znajdować się w odpowiedzi. Każdorazowo wysyłane zostają 3 bajty: echo kodu zapytania – w celu identyfikacji przez moduł MASTER skąd nadeszła odpowiedź oraz 2 bajty zawierających przesyłaną wartość. Dane zbierane przez sensor są typu float (wartości zmiennoprzecinkowe, 32-bitowe), a biblioteka może obsługiwać tylko tablice, gdzie każde pole należy do typu liczb całkowitych o stałej szerokości 8 bitów (`uint8_t`, reprezentuje wartości tylko dodatnie o wartości nie większej niż 255 [1]). Aby móc swobodnie przesyłać wartości zastosowany został rzut (ang. *type casting*) do oryginalnej wielkości do typu `uint16_t` oraz podział jej na dwie części, wykorzystując do tego maski oraz operacje bitowe. Tak przygotowana tablica wysłana jest jako odpowiedź do modułu MASTER.

4.3 Implementacja oprogramowania modułu serwera sieciowego

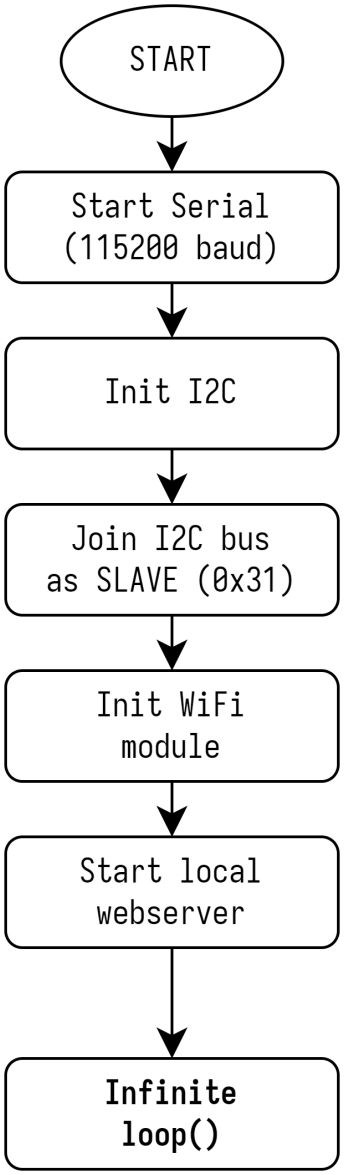
Oprogramowanie modułu serwera sieciowego dzieli się na dwie części: implementację działania serwera sieciowego (ang. *webserver*) oraz odbierania i dekodowania informacji z modułu MASTER sieci LoRa. Tak jak w przypadku oprogramowania dla modułów sieci LoRa, w przypadku modułu serwera sieciowego na początku jednorazowo wykonywana jest seria instrukcji zawartych w sekcji `loop()`, a następnie przechodzi do wykonywania instrukcji w nieskończonej pętli. Schemat działania tej części kodu przedstawia diagram blokowy na rys. 8, natomiast na listingu 21 przedstawiona została implementacja tej części oprogramowania.

```
void setup()
{
    Wire.begin(I2C_ADDR);
    Wire.onReceive(i2cReceive);
    Serial.begin(115200);

    wifi::init();
    webserver::init();
}
```

Listing 21. Implementacja funkcji `setup()` modułu serwera sieciowego

Pierwszym elementem jest ustawienie port szeregowy na 115200 baud (szybkość transmisji), a następnie inicjalizacja magistrali I2C oraz dołączenie modułu w roli SLAVE (odbiorca). Ostatnimi krokami wykonywanymi przed przejściem do nieskończonej pętli jest, wykorzystując zaimplementowane funkcje `wifi::init()` oraz `webserver::init()`, inicjalizacja modułu WiFi oraz samego serwera sieci lokalnej. Na listingach 22 oraz 23 przedstawione zostały kody źródłowe tych funkcji.



Rysunek 8. Diagram blokowy sekcji `setup()` oprogramowania modułu serwera sieciowego

```
    void init(void)
    {
        debug::println(debug::INFO, "Trying to init WiFi module...");
        WiFi.setPins(WIFISHIELD_PINS);

        // Check if WiFi module is present
        if (WiFi.status() == WL_NO_SHIELD)
        {
            debug::println(debug::ERR, "WiFi module init fail (disconnected
                ?)");
            while (true)
            {
                // Pause forever
            }
        }

        pinMode(LED_BUILTIN, OUTPUT);
        debug::println(debug::INFO, "Attempting connection to: " + String(
            WIFI_SSID));

        while (WiFi.status() != WL_CONNECTED)
        {
            WiFi.begin(WIFI_SSID, WIFI_PASS);

            digitalWrite(LED_BUILTIN, 1);
            delay(250);
            digitalWrite(LED_BUILTIN, 0);
            delay(250);
        }

        digitalWrite(LED_BUILTIN, 1);
        printStatus();
    }
```

Listing 22. Implementacja funkcji `wifi::init()`

```
    WiFiServer server(SERVER_PORT);
    void init(void)
    {
        while (WiFi.status() != WL_CONNECTED)
        {
            // Wait for WiFi to connect before starting server
        }

        debug::println(debug::INFO, "Webserver started on port " + String(
            SERVER_PORT, DEC));
        server.begin();
    }
```

Listing 23. Implementacja funkcji `webserver::init()`

Inicjalizacja WiFi wykorzystuje zdefiniowane w osobnym pliku konfiguracyjnym SSID (nazwę) oraz hasło do sieci WiFi, gdzie moduł ma wykonać próbę podłączenia. Do implementacji funkcji została dodana także funkcja sprawdzenia czy rozszerzenie WiFi na płytce Feather jest poprawnie podłączone. Gdy moduł serwera próbuje zalogować się do sieci, wbudowana dioda miga, natomiast, gdy połączenie zostanie uzyskane świeci światłem stałym. Ponieważ do poprawnego działania wymagane jest połączenie z siecią WiFi, weryfikacja tego kroku została zaimplementowana w postaci nieskończonej pętli while, która blokuje dalsze działanie oprogramowania, jeżeli modułowi nie został przydzielony adres IP w sieci lokalnej. Implementacja oraz działanie funkcji inicjalizacji serwera sieciowego jest znacznie mniej skomplikowane – tworzony jest nowy obiekt serwera `server`, następnie kod oczekuje na połączenie z siecią WiFi, po czym serwer zostaje uruchomiony i można się do niego podłączyć korzystając z logowanego przez port szeregowy adresu IP. Na rys. 9 przedstawiony został zrzut ekranu ze strony internetowej dostępnej w sieci lokalnej, gdzie podłączony został moduł serwera sieciowego.

LoRa Private sensor network

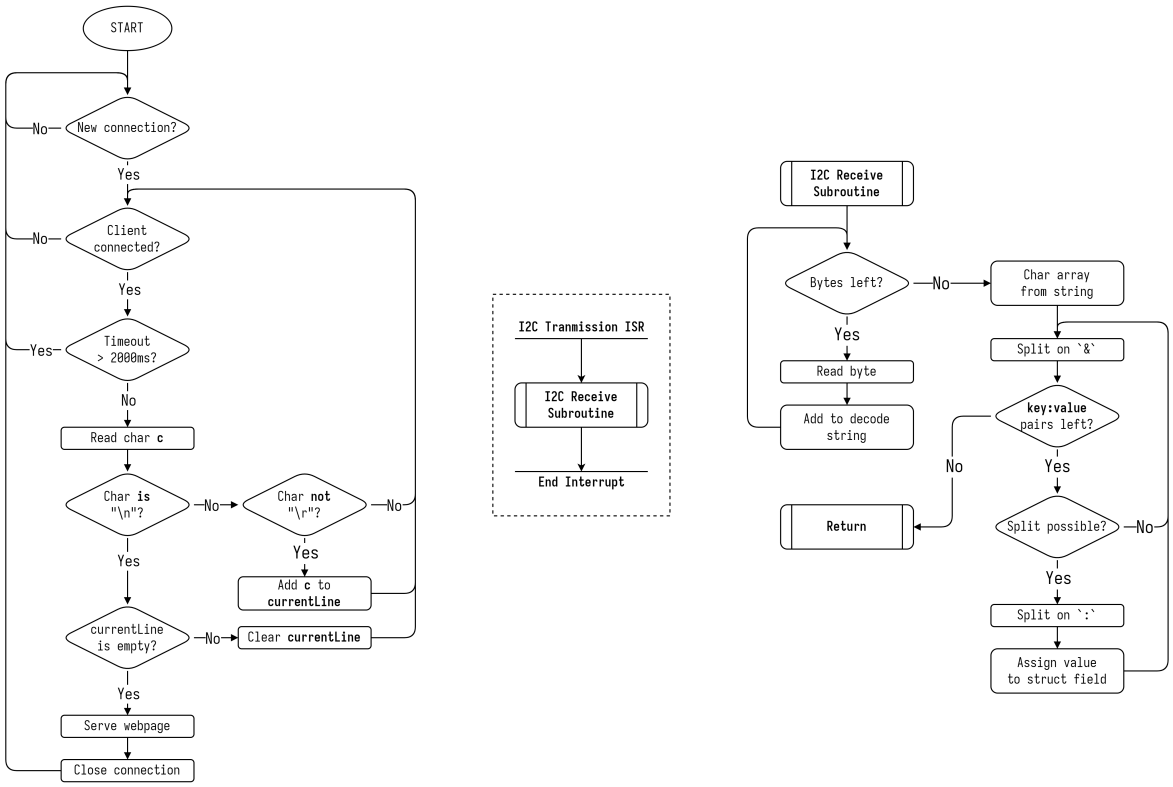
MEASUREMENT	NODE 1	NODE 2	NODE 3
Temperature [°C]	24.4	25.5	25.0
Pressure [hPa]	1001.0	1002.0	1001.0
Humidity [%]	55.8	52.5	54.8
Failed req. [%]	4.2	8.3	66.7

Rysunek 9. Wygląd strony internetowej zaimplementowanej do wizualizacji danych zbieranych z sieci

Po zakończonej inicjalizacji magistrali I2C, skonfigurowaniu oraz podłączeniu się do WiFi oraz uruchomieniu serwera oprogramowanie przechodzi do nieskończonej pętli. Zaimplementowane zostały tutaj funkcje odpowiedzialne za serwowanie strony internetowej dla użytkownika oraz odbieranie danych przez magistralę I2C od modułu MASTER sieci LoRa oraz dekodowanie ich. Na rys. 10 przedstawiony został schemat blokowy działania obu tych funkcji.

4.3.1 Wyświetlanie strony internetowej z danymi

Implementacja serwera sieciowego oraz strony, do której użytkownik może uzyskać dostęp w sieci lokalnej bazuje na bibliotece WiFi101. W każdej iteracji pętli sprawdzane jest czy do serwera próbuje



Rysunek 10. Schemat blokow działania funkcji w nieskończonej pętli serwera sieciowego

podłączyć się nowy klient. Jeżeli otrzymane zostanie nowe żądanie (ang. *request*), to inicjalizowany jest timer sprawdzający czy nie nastąpiło przekroczenie na wysłanie odpowiedzi lub klient nie stracił połączenia. Wartość ta została przyjęta jako 2 sekundy – wartość wystarczająco duża, aby połączenie nie było zamykane zbyt wcześnie oraz wystarczająco niska, aby inne żądania mogły zostać obsłużone bez nadmiernego oczekiwania. Kod źródłowy tej części przedstawiony został na listingu 24.

```
1     wifi::client = webserver::server.available();
2     if (wifi::client)
3     {
4         timeoutTimer = millis();
5         String currentLine = "";
6
7         while (wifi::client.connected() &&
8                (millis() - timeoutTimer) <= CLIENT_TIMEOUT)
9         {
10            /**
11             * Process request from the connecting client
12             */
13        }
14        webserver::header = "";
15        wifi::client.stop();
16    }
```

Listing 24. Implementacja obsługi nowego żądania w oprogramowaniu serwera sieciowego

W momencie otrzymania nowego żądania serwer odbiera pojedynczo wszystkie przesyłane do niego znaki. Przy każdym znaku sprawdzane jest czy nie nastąpiło rozłączenie z klientem. Odbierane znaki są sprawdzane czy jest to znak nowej linii (\n) oraz czy obecnie następna linia jest pusta – jest to sygnał, że całe zapytanie zostało odebrane, a serwer powinien wysłać odpowiedź. W przypadku, gdy znakiem nie jest \n, ani \r zostaje on dodany do obecnej linii, tak aby zebrać pełny nagłówek żądania, zawierający wszystkie informacje. Implementacja tego elementu przedstawiona została na listingu 25.

```
            char c = wifi::client.read();
            webserver::header += c;

            if (c == '\n')
            {
                if (currentLine.length() == 0)
                {
                    webserver::serve(&storedData);
                    break;
                }
                currentLine = "";
            }
            else if (c != '\r')
            {
                currentLine += c;
            }
```

Listing 25. Implementacja odbierania przychodzącego żądania

Przesyłanie odpowiedzi przez serwer wykonywane wykorzystując do tego zaimplementowaną funkcję `webserver::serve()`. Jej implementacja przedstwiona została na listingu 26. W celu przesłania odpowiedzi na żądanie oprócz zawartości samej strony wymagane jest przesłanie także nagłówka strony. Aby zmniejszyć ilość wymaganej pamięci SRAM wymaganej na oprogramowanie znaczna część kodu strony (nagłówek HTTP, nagłówek strony `<head>` oraz część wyświetlanej użytkownikowi strony) przechowywana jest w pamięci programowej `PROGMEM` (pamięci flash) mikrokontrolera. Elementy zostają pobrane ze zdefiniowanych zmiennych oraz przesłane do żądającego klienta wykorzysując do tego pętle typu `for-range`. Do przesłania elementów, gdzie wymagane są dane, które moduł odebrał z sieci LoRa, wykorzystana została zaimplementowana funkcja szablonowa `printRow` (przesyłanie jednego wiersza tabeli z danymi). Kod źródłowy tej funkcji przedstawiony został na listingu 27.

```
void serve(StoredData_t *data)
{
    for (String line : httpHeader)
    {
        wifi::client.println(line);
    }
    wifi::client.println();

    for (String line : webpageHead)
    {
        wifi::client.println(line);
    }

    for (u8 i = 0; i < ARRAYSIZE(webpageBody); ++i)
    {
        if (i == BODY_TAB_IDX)
        {
            printRow(data->temperature, "Temperature [&deg;C]");
            printRow(data->pressure, "Pressure [hPa]");
            printRow(data->humidity, "Humidity [%]");
            printRow(data->failedPercent, "Failed req. [%]");
            continue; // Skip printing from array
        }
        wifi::client.println(webpageBody[i]);
    }
}
```

Listing 26. Implementacja funkcji do przesyłania zawartości strony internetowej `webserver::serve()`

```
template <typename T, size_t size, size_t length>
void printRow(const T (&array)[size], const char (&rowTitleStr)[length
    ])
{
    static u8 fmtCharCount = 15;
    char line[length + fmtCharCount];

    sprintf(line, "<tr><td>%s</td>", rowTitleStr);
    wifi::client.println(line);

    // Template type does not matter, values cast to f32 for display
    for (T value : array)
    {
        char cell[25];
        sprintf(cell, "<td>%.1f</td>", (f32)value);
        wifi::client.println(cell);
    }

    wifi::client.println("</tr>");
}
```

Listing 27. Implementacja funkcji szablonowej `printRow()`

Funkcja opiera się na modyfikacji ciągu znaków zawierającego format pojedynczej komórki tabeli oraz pętli typu `for-range`. Wykorzystuje do tego funkcję `sprintf`, przyjmując jako argument ciąg, który ma zostać w to miejsce wstawiony – nagłówek wiersza (ciąg znaków, tablicę `const char`), który zawiera informację o typie danych w danym wierszu lub bezpośrednią wartość otrzymaną z sieci LoRa (wartość zmiennoprzecinkowa, `float`).

4.3.2 Odbieranie oraz dekodowanie danych z sieci LoRa

W celu uzyskania jak najlepszej wydajności odbierania oraz dekodowania otrzywanych danych zaimplementowane rozwiązanie w znacznym stopniu ogranicza wykorzystanie funkcji, która powoduje dynamiczny przydział pamięci (ang. *dynamic memory allocation*). Pierwszym elementem jest odebranie pełnej wiadomości. Aby przetwarzanie danych było łatwiejsze każdy, pojedynczy znak jest dodawany do ciągu znaków (string) – jest to jedyny element implementacji wykorzystujący dynamiczną alokację. Odpowiedzialny za to fragment kodu przedstawiony został na listingu 28.

```
void i2cReceive(int byteCount)
{
    // Build a string from I2C transmission
    String recv;
    while (Wire.available() > 0)
    {
        char recvChar = Wire.read();
        recv += recvChar;
    }

    // Decode into values
    decodeStr(&recv, &storedData);
}
```

Listing 28. Implementacja funkcji odbierającej dane przez magistralę I2C

Następnie wywoływana jest funkcja `decodeStr()`, której kod przedstawiony został na listingu 29. Pierwszym elementem jest przetworzenie wejściowego ciągu znaków na tablicę pojedynczych znaków o długości większej o jeden znak. Dodatkowy element potrzeby jest dla algorytmu w celu oznaczenia zakończenia tablicy. Dodatkowo definiowana jest tablica czteroelementowa do przechowywania indeksu obecnie uzpełnianej danej `dataCounter`. Wykorzystując funkcję z języka C `strtok()` tablica dzielona jest na znaku „&" – wykorzystanego jako łączenie pomiędzy parami klucz-wartość. Drugim elementem jest podział każdej pary na znaku „:" – łącznika pomiędzy kluczem (od 0 do 4) oraz wartością. Każdorazowo algorytm sprawdza klucz i na jego podstawie otrzymana dana jest przypisywana do odpowiedniego pola struktury na indeksie, którego wartość śledzi licznik (`dataCounter`). Wszystkie operacje są następnie powtarzane do momentu, gdy algorym dotrze do końca tablicy.

```
void decodeStr(String *recv, StoredData_t *data)
{
    char recvArray[recv->length() + 1];
    recv->toCharArray(recvArray, recv->length());
    recvArray[recv->length()] = 0;

    // Keeps count for each struct member
    u8 dataCounter[4] = {0, 0, 0, 0};

    /**
     * Read key-value pairs
     * Split string on `&` -> each key-value pair
     * Split pairs on `:` -> key : value
     */
    char *keyToken = strtok(recvArray, "&");
    while (keyToken != NULL)
    {
        char *valueToken = strchr(keyToken, ':');
        if (valueToken != 0)
        {
            *valueToken = 0;
            int key = atoi(keyToken);
            ++valueToken;

            // Assign value from valueToken to correct struct member
            switch (key)
            {
            case TEMPERATURE:
                data->temperature[dataCounter[TEMPERATURE]++] =
                    (f32)atoi(valueToken) / 100.0f;
                break;
            case PRESSURE:
                data->pressure[dataCounter[PRESSURE]++] = (f32)atoi(
                   valueToken);
                break;
            case HUMIDITY:
                data->humidity[dataCounter[HUMIDITY]++] =
                    (f32)atoi(valueToken) / 100.0f;
                break;
            case FAILPERCENT:
                data->failedPercent[dataCounter[FAILPERCENT]++] =
                    (f32)atoi(valueToken) / 100.0f;
                break;
            }
        }
        keyToken = strtok(NULL, "&");
    }
}
```

Listing 29. Implementacja funkcji do dekodowania danych przesyłanych przez moduł MASTER

Rozdział 5

Podstawowe testy implementacji

Rozdział 6

Badania działającej sieci

Rozdział 7

Podsumowanie

Summary

Bibliografia

[1] cppreference.com, Fixed width integer types.

[2] Doxygen manual, Documenting the code.

[3] PlatformIO Documentation, Development Platforms.

[4] Snyk, Open source licenses: What is a software license?

[5] STM32duino repository, Arduino core for STM32 MCUs.

Spis rysunków