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

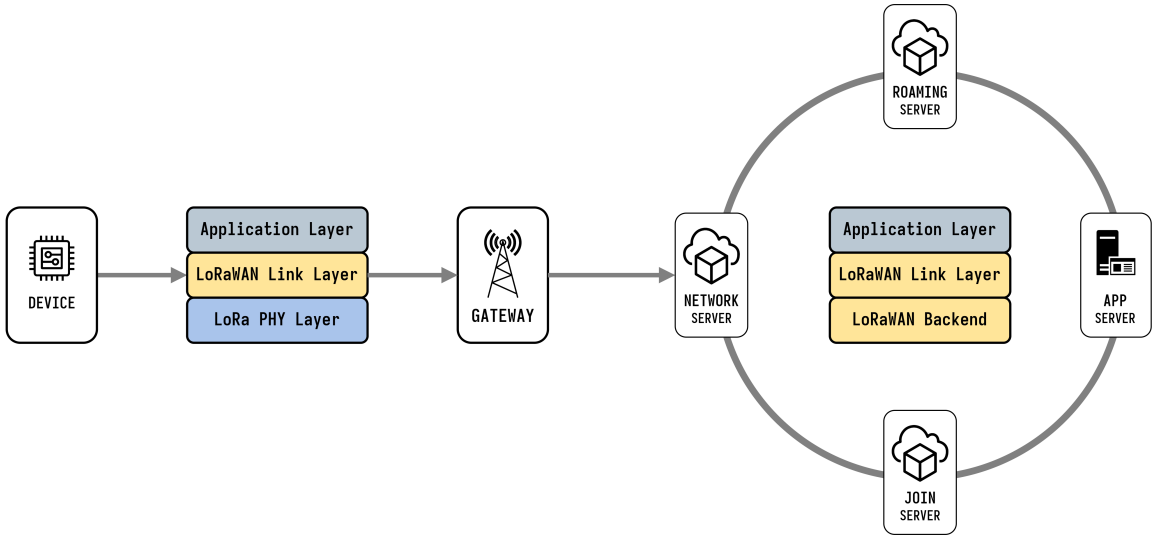
Wstęp

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 [2], 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.

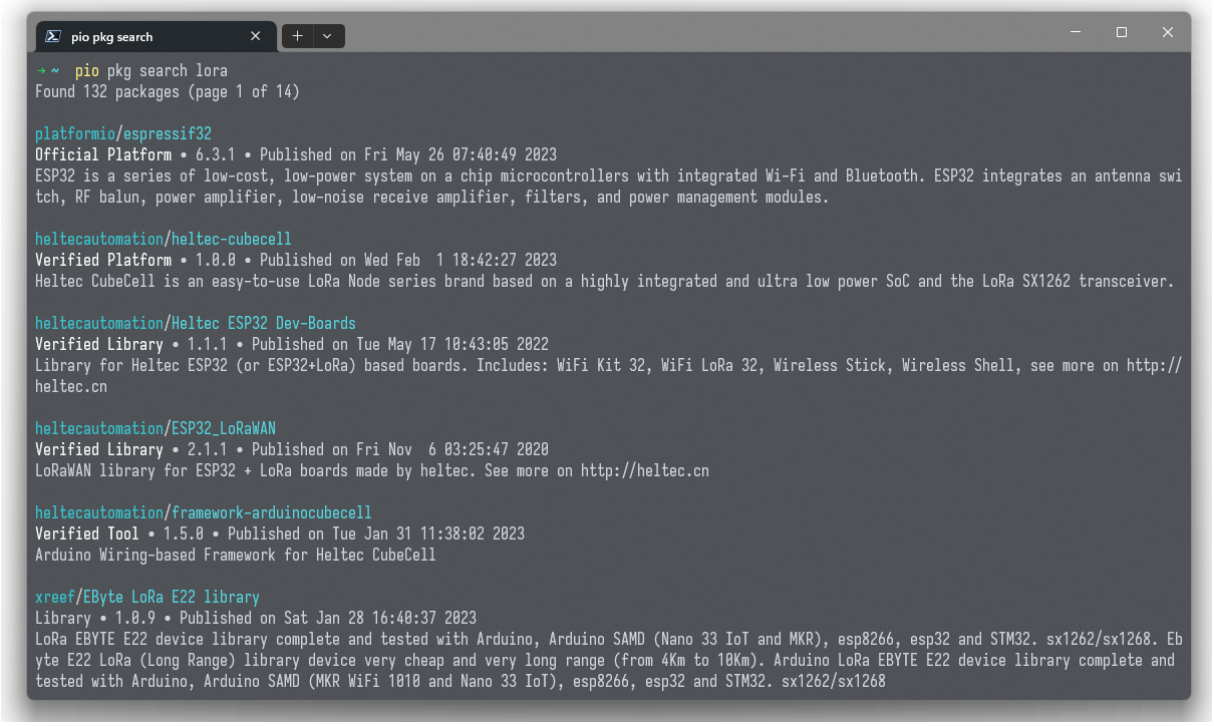
W przypadku opcji związanych z portem, jeżeli nie zostaną sprecyzowane (podane jako argument do komendy), PlatformIO będzie próbował wykryć je automatycznie. Dostępne jest jeszcze kilka innych opcji, jednkże są one znacznie rzadziej wykorzystywane, ponieważ ich domyślne opcje są tymi, które są najczęściej 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*). Dodatkowo, ponieważ kod źródłowy jest dostępny w domenie publicznej (repozytorium na platformie GitHub z licencją *MIT* [3]), 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 [1].

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 [4]. 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).

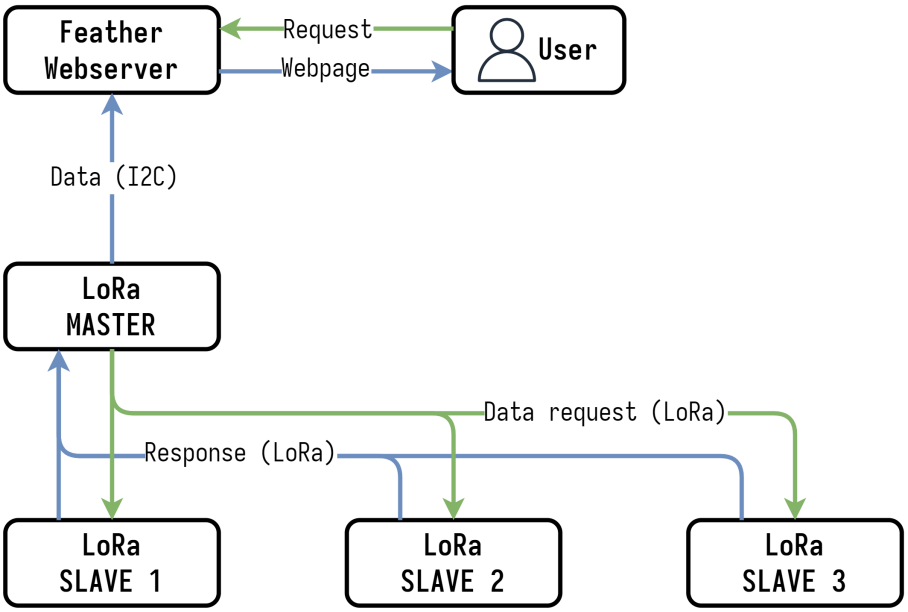
Ponadto, wykorzystane zostały biblioteki I2C oraz SPI, dostępne w rdzeniu Arduino. Potrzebne były one do uzyskania komunikacji pomiędzy mikrokontrolerem Adafruit Feather M0 a modułem WiFi, sensorami BM280 podłączonymi do modułów SLAVE oraz do stworzenia połączenia pomiędzy modułem MASTER a płytką z serwerem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

```
1 [env:SLAVE1]
2 extends = base_config
3 build_flags =
4   -DBOARD_TYPE=lora::SLAVE
```

```
5 -DBOARD_ID=0x01
```

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

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 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:

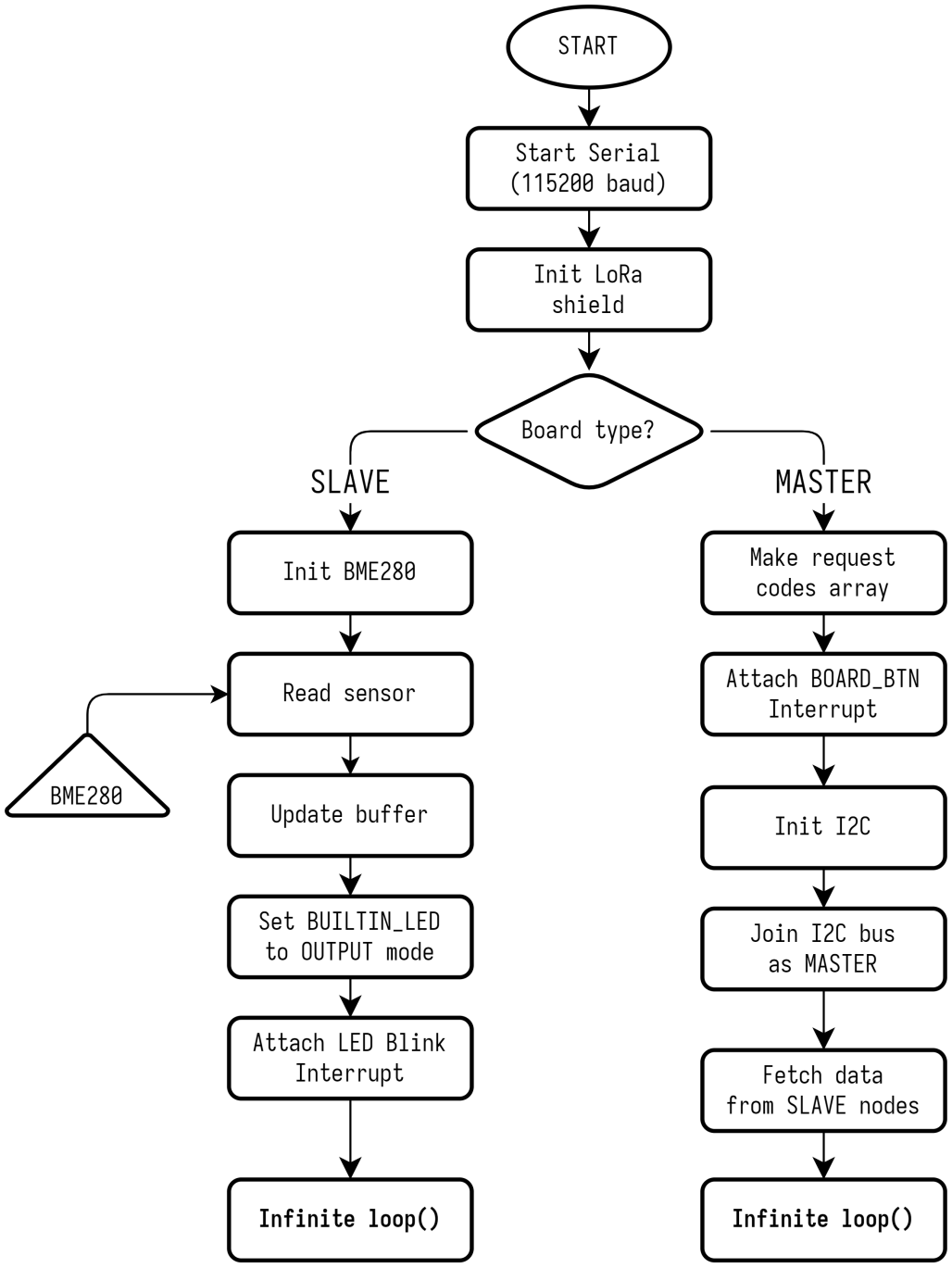
1. przygotowanie sensora BME280 oraz pobranie z niego danych,
2. aktualizacja zawartości bufora (wykorzystywanego do przechowywania odczytanych wartości),
3. przygotowanie diody LED, która informuje o trwającej komunikacji w sieci,
4. przygotowanie przerwania, wykorzystywanego do obsługi nowych zapytań.

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.

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



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

```
#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);
```

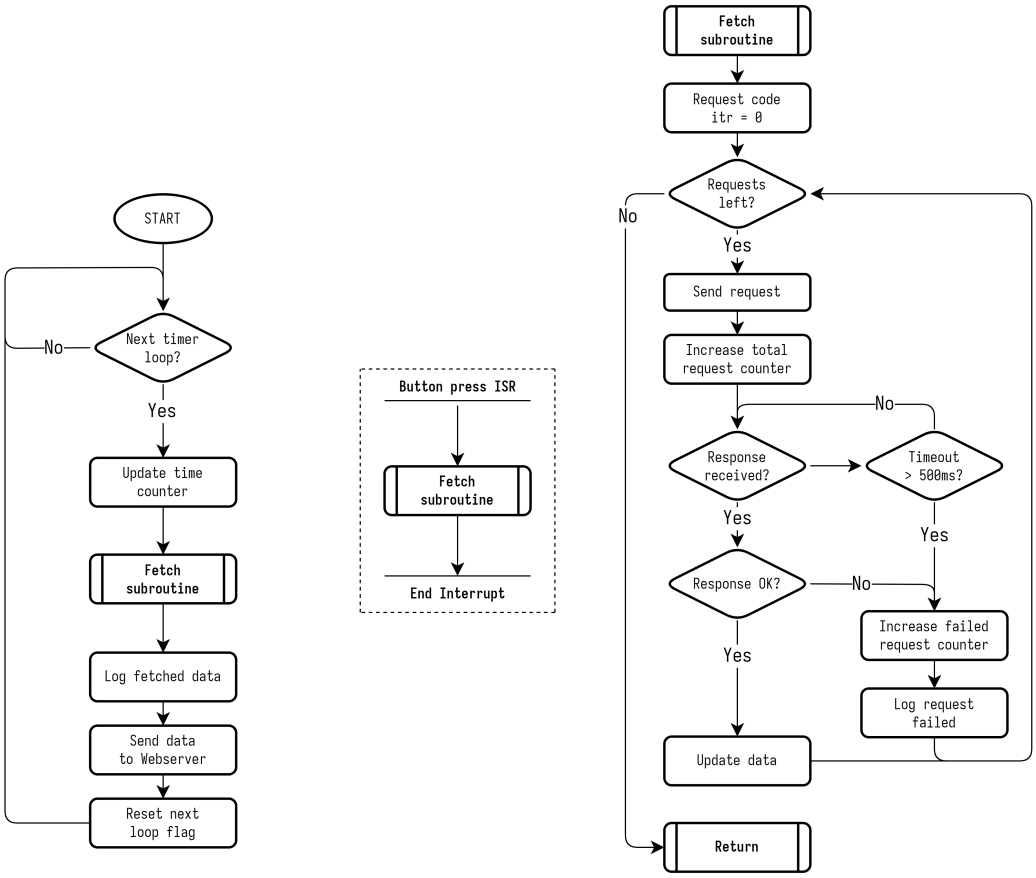
Listing 3. Funkcji wykorzystywana do wysyłania sformatowanych wiadomości przez port szeregowy

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.



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

```
    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

```
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

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, która dostępna jest w bibliotece do obsługi modułu rozszerzeń LoRa, wysyłana jest wiadmość 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 wystapił żaden z tych 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 wykorzytanie 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 listingu 8, natomiast implementacja pełnych funkcji do przesyłania danych na listingu 9. Dodatkowo zaimplementowana została także funkcja pomocniczna 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 10.

```
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();
}
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 8. Zaimplementowane funkcje szablonowe `logValues()` oraz `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();
}

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 9. Funkcje wykorzystywane do logowania wartości przez port szeregowy oraz 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)
```

```
5       {
6           failedPercent[i] = ((f32)failReq[i] / (f32)totalReq[i]) * 100.0f;
7       }
8   }
```

Listing 10. Implementacja funkcji pomocniczej do wyznacznia wartości procentowej zapytań z błędem

4.2.2 Oprogramowanie modułów SLAVE

4.3 Implementacja oprogramowania modułu serwera sieciowego

Rozdział 5

Podstawowe testy implementacji

Rozdział 6

Badania działającej sieci

Rozdział 7

Podsumowanie

Summary

Bibliografia

[1] Doxygen, *Doxygen manual*.

[2] PlatformIO, *Documentation*, Development Platforms.

[3] Snyk, What is a software license?

[4] STM32duino, *Documentation*, Arduino core for STM32 MCUs.

Spis rysunków