

Politechnika Warszawska

W Y D Z I A Ł E L E K T R Y C Z N Y



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

1	Wstęp	9
2	Sieci w standardzie LoRa	11
2.1	LoRaWAN	11
3	Przygotowanie środowiska programistycznego	13
3.1	Rozpoczęcie projektu z PlatformIO Core	13
3.2	Praca z PlatformIO	14
3.2.1	Uruchamianie projektu	15
3.2.2	Zarządzanie bibliotekami	16
4	Implementacja oprogramowania	19
4.1	Framework oraz biblioteki	19
4.1.1	Wykorzystane biblioteki	20
4.1.2	Ograniczenia związane z wykorzystaniem Arduino oraz STM32duino	20
4.2	Implementacja oprogramowania elementów sieci	21
4.2.1	Oprogramowanie modułu MASTER	25
4.2.2	Oprogramowanie modułów SLAVE	32
4.3	Implementacja oprogramowania modułu serwera sieciowego	38
4.3.1	Wyświetlanie strony internetowej z danymi	43
4.3.2	Odbieranie oraz dekodowanie danych z sieci LoRa	46
5	Testy implementacji	49
5.1	Testy komunikacji sieci	49
5.2	Testy transmisji danych do serwera sieciowego	51
5.3	Testy działania serwera sieciowego oraz strony internetowej	52
6	Badania działającej sieci	55
6.1	Badanie komunikacji w sieci	55
7	Podsumowanie	57
	Bibliografia	59

Rozdział 1

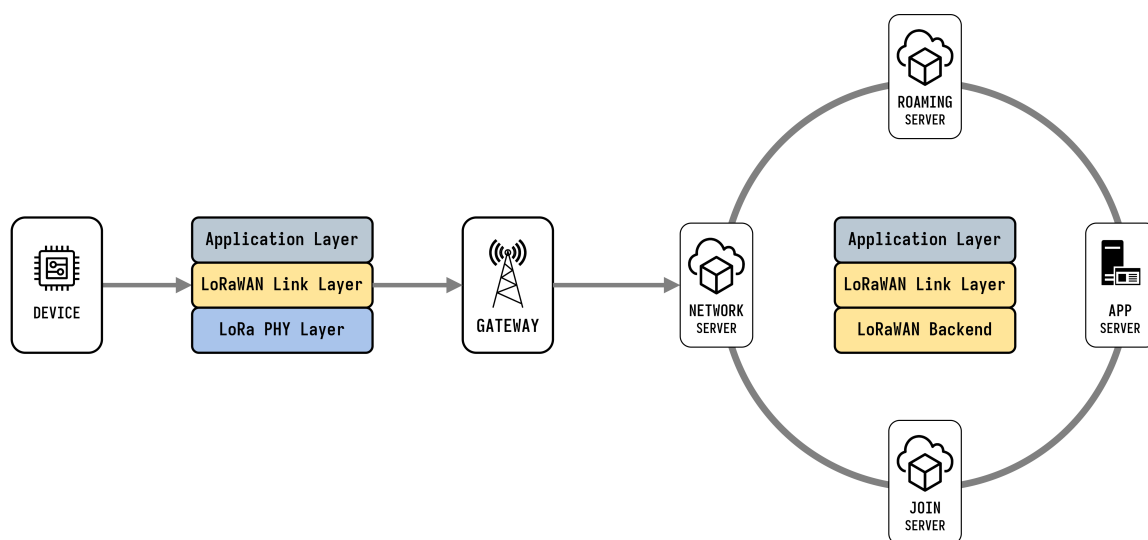
Wstęp

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”.

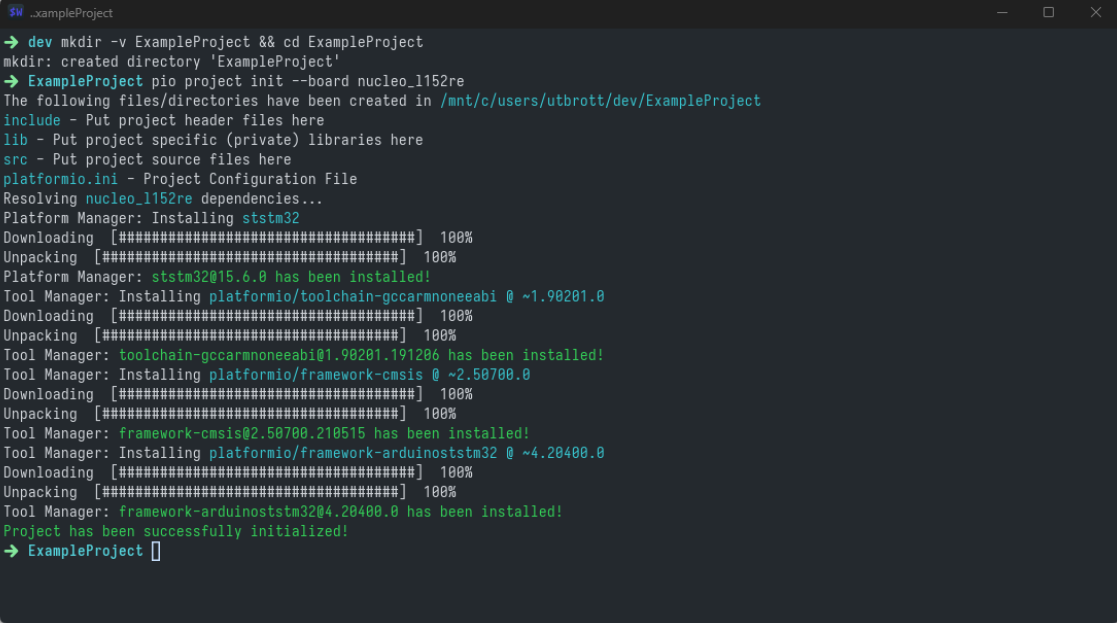
Rozwiązanie to zostało wybrane jako główne narzędzie do kompilacji oraz wgrywania kodu źródłowego, z uwagi na to, że działa na wielu platformach. Dzięki temu nie jest wymagane instalowanie oraz ustawianie osobnych, dedykowanych środowisk dla każdej z wykorzystywanych platform. Jedynym wymogiem, aby móc 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>
```

Zrzut ekranu z procesu tworzenia przykładowego projektu, wykorzystując do tego PlatformIO Core przedstawiony został na rys. 2. Widoczny proces instalacji dodatkowych narzędzi wymagany jest zawsze przy tworzeniu nowego projektu na nowej platformie (w przypadku, gdy platforma była już wykorzystywana PlatformIO weryfikuje tylko czy zainstalowane biblioteki, narzędzia są w najnowszej wersji). Ponieważ komenda została wywołana wewnątrz maszyny wirtualnej, na świeżo zainstalowanym PlatformIO wymagane było pobranie oraz instalacja potrzebnych elementów.



```

ExampleProject
→ dev mkdir -v ExampleProject && cd ExampleProject
mkdir: created directory 'ExampleProject'
→ ExampleProject pio project init --board nucleo_l152re
The following files/directories have been created in /mnt/c/users/utbrott/dev/ExampleProject
include - Put project header files here
lib - Put project specific (private) libraries here
src - Put project source files here
platformio.ini - Project Configuration File
Resolving nucleo_l152re dependencies...
Platform Manager: Installing ststm32
Downloading [#####] 100%
Unpacking [#####] 100%
Platform Manager: ststm32@15.6.0 has been installed!
Tool Manager: Installing platformio/toolchain-gccarmnoneabi @ ~1.90201.0
Downloading [#####] 100%
Unpacking [#####] 100%
Tool Manager: toolchain-gccarmnoneabi@1.90201.191206 has been installed!
Tool Manager: Installing platformio/framework-cmsis @ ~2.50700.0
Downloading [#####] 100%
Unpacking [#####] 100%
Tool Manager: framework-cmsis@2.50700.210515 has been installed!
Tool Manager: Installing platformio/framework-arduino-ststm32 @ ~4.20400.0
Downloading [#####] 100%
Unpacking [#####] 100%
Tool Manager: framework-arduino-ststm32@4.20400.0 has been installed!
Project has been successfully initialized!
→ ExampleProject []

```

Rysunek 2. Tworzenie nowego projektu wykorzystując do tego PlatformIO Core (CLI)

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 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 zostać wgrany kod. Szczególnie użyteczne w przypadku, gdy pracuje się na wielu urządzeniach (mikrokontrolerach) 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, jednakże są one znacznie rzadziej wykorzystywane, ponieważ ich domyślne opcje są tymi, które są najczęściej ustawiane. Przykładowe wykorzystanie komendy do kompilacji projektu przedstawione zostało na rys. 3.

```

PS pwsh.exe
+ ~\Dev\lora-psn git:(main) pio run
Processing MASTER (platform: ststm32; board: nucleo_l152re; framework: arduino)
-----
Verbose mode can be enabled via '-v, --verbose' option
CONFIGURATION: https://docs.platformio.org/page/boards/ststm32/nucleo_l152re.html
PLATFORM: ST STM32 (15.6.0) > ST Nucleo L152RE
HARDWARE: STM32L152RET6 32MHz, 80KB RAM, 512KB Flash
DEBUG: Current (stlink) On-board (stlink) External (blackmagic, cmsis-dap, jlink)
PACKAGES:
- framework-arduino-ststm32 @ 4.20400.0 (2.4.0)
- framework-cmsis @ 2.50700.210515 (5.7.0)
- toolchain-gccarm-none-eabi @ 1.90201.191206 (9.2.1)
LDF: Library Dependency Finder -> https://bit.ly/configure-pio-ldf
LDF Modes: Finder ~ chain, Compatibility ~ soft
Found 16 compatible libraries
Scanning dependencies...
Dependency Graph
|-- Wire @ 1.0.0
|-- SPI @ 1.0.0
|-- STM32duino I-NUCLEO-LRWAN1 @ 1.0.1
|-- Adafruit BME280 Library @ 2.2.2
|-- Adafruit BusIO @ 1.14.1
|-- Adafruit Unified Sensor @ 1.1.9
Building in release mode
Checking size .pio\build\MASTER\firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM: [ ] 2.5% (used 2064 bytes from 81920 bytes)
Flash: [= ] 7.2% (used 37756 bytes from 524288 bytes)
===== [SUCCESS] Took 2.18 seconds =====

```

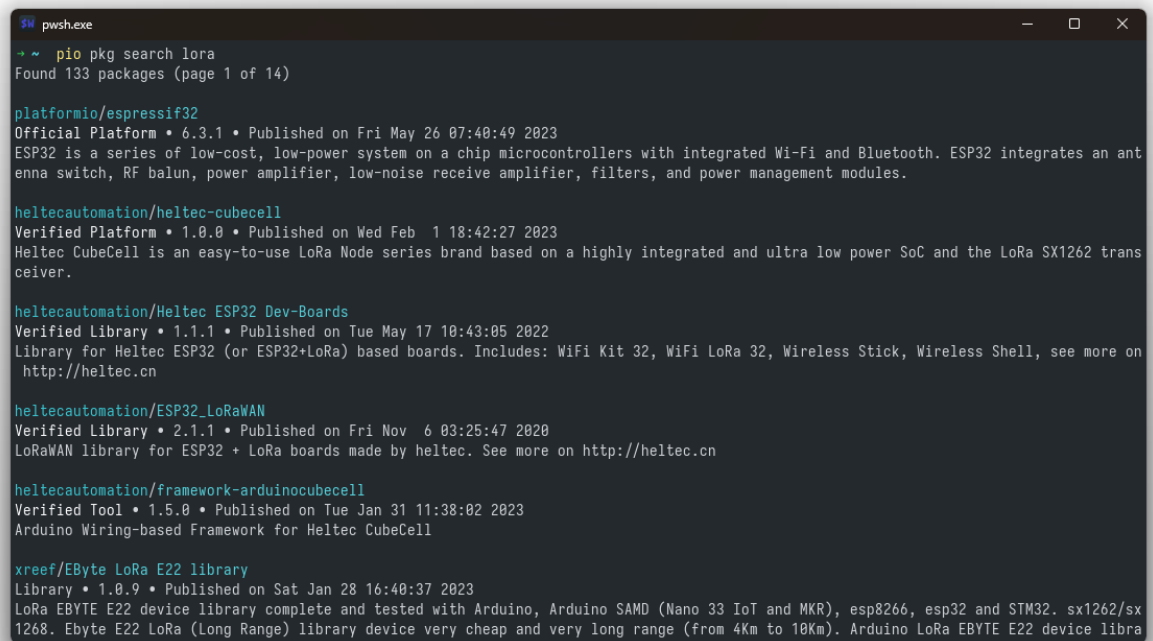
Rysunek 3. Przykładowe wywołanie komendy `pio run`, tutaj do kompilacji projektu

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, 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. Wykorzystując 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. 4 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. Instalowanie bibliotek odbywa się poprzez wykorzystanie argumentów `install --library` oraz podanie nazwy oraz wersji wybranego elementu (np. z uprzednio wyszukanej listy). Przykładowe wykorzystanie tej komendy przedstawione zostało na rys. 5. Podczas instalacji (co widoczne jest na zrzucie ekranu) narzędzie weryfikuje – oraz jeżeli jest to potrzebne instaluje – jakie inne biblioteki są wymagane przez tą, którą użytkownik dodaje do projektu.



```

$W pwsh.exe
→ ~ pio pkg search lora
Found 133 packages (page 1 of 14)

platformio/espressif32
Official Platform • 6.3.1 • Published on Fri May 26 07:40:49 2023
ESP32 is a series of low-cost, low-power system on a chip microcontrollers with integrated Wi-Fi and Bluetooth. ESP32 integrates an antenna switch, RF balun, power amplifier, low-noise receive amplifier, filters, and power management modules.

heltecautomation/heltec-cubecell
Verified Platform • 1.0.0 • Published on Wed Feb 1 18:42:27 2023
Heltec CubeCell is an easy-to-use LoRa Node series brand based on a highly integrated and ultra low power SoC and the LoRa SX1262 transceiver.

heltecautomation/Heltec ESP32 Dev-Boards
Verified Library • 1.1.1 • Published on Tue May 17 10:43:05 2022
Library for Heltec ESP32 (or ESP32+LoRa) based boards. Includes: WiFi Kit 32, WiFi LoRa 32, Wireless Stick, Wireless Shell, see more on http://heltec.cn

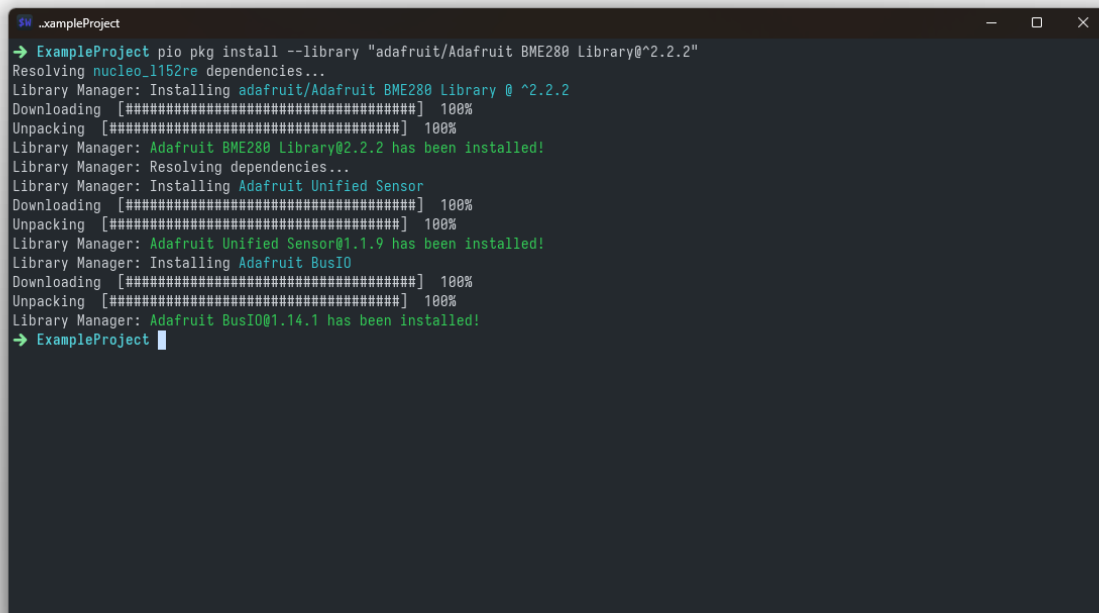
heltecautomation/ESP32 LoRaWAN
Verified Library • 2.1.1 • Published on Fri Nov 6 03:25:47 2020
LoRaWAN library for ESP32 + LoRa boards made by heltec. See more on http://heltec.cn

heltecautomation/framework-arduino-cubecell
Verified Tool • 1.5.0 • Published on Tue Jan 31 11:38:02 2023
Arduino Wiring-based Framework for Heltec CubeCell

xreef/EByte LoRa E22 library
Library • 1.0.9 • Published on Sat Jan 28 16:40:37 2023
LoRa EBYTE E22 device library complete and tested with Arduino, Arduino SAMD (Nano 33 IoT and MKR), esp8266, esp32 and STM32. sx1262/sx1268. Ebyte E22 LoRa (Long Range) library device very cheap and very long range (from 4Km to 10Km). Arduino LoRa EBYTE E22 device library

```

Rysunek 4. Wyniki wyszukiwania bibliotek powiązanych z hasłem „LoRa”



```

$W _xampleProject
→ ExampleProject pio pkg install --library "adafruit/Adafruit BME280 Library@^2.2.2"
Resolving nucleo_l152re dependencies...
Library Manager: Installing adafruit/Adafruit BME280 Library @ ^2.2.2
Downloading [#####] 100%
Unpacking [#####] 100%
Library Manager: Adafruit BME280 Library@2.2.2 has been installed!
Library Manager: Resolving dependencies...
Library Manager: Installing Adafruit Unified Sensor
Downloading [#####] 100%
Unpacking [#####] 100%
Library Manager: Adafruit Unified Sensor@1.1.9 has been installed!
Library Manager: Installing Adafruit BusIO
Downloading [#####] 100%
Unpacking [#####] 100%
Library Manager: Adafruit BusIO@1.14.1 has been installed!
→ ExampleProject

```

Rysunek 5. Przykład wykorzystania komendy do instalacji bibliotek w projekcie

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).

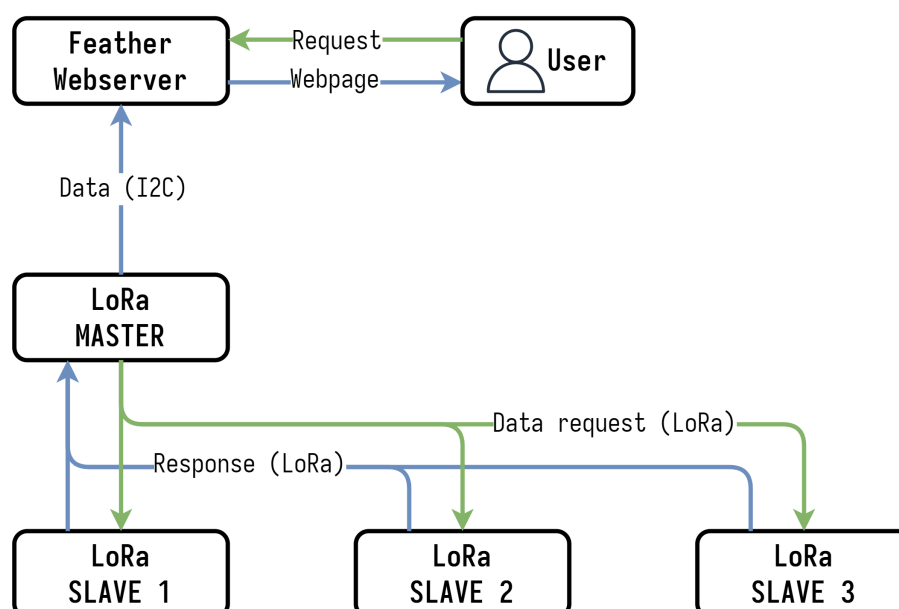
Ponadto, wykorzystane zostały biblioteki I2C oraz SPI, dostępne w rdzeniu Arduino. Potrzebne były one do uzyskania komunikacji pomiędzy mikrokontrolerem Adafruit Feather M0 a modułem WiFi, sensorami BM280 podłączonymi do modułów SLAVE oraz do stworzenia połączenia pomiędzy modułem MASTER a płytką z serwerem 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 przerw programowych oraz ograniczone możliwości zastosowania przerw 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 podrozdziale 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. 6.



Rysunek 6. 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

```
1 [env:SLAVE1]
2 extends = base_config
3 build_flags =
4     -DBOARD_TYPE=lorax:SLAVE
5     -DBOARD_ID=0x01
```

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

odpowiednio raz, podczas startu modułu oraz w nieskończonej pętli, dopóki płytkę ma zasilanie. Na rys. 7 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:

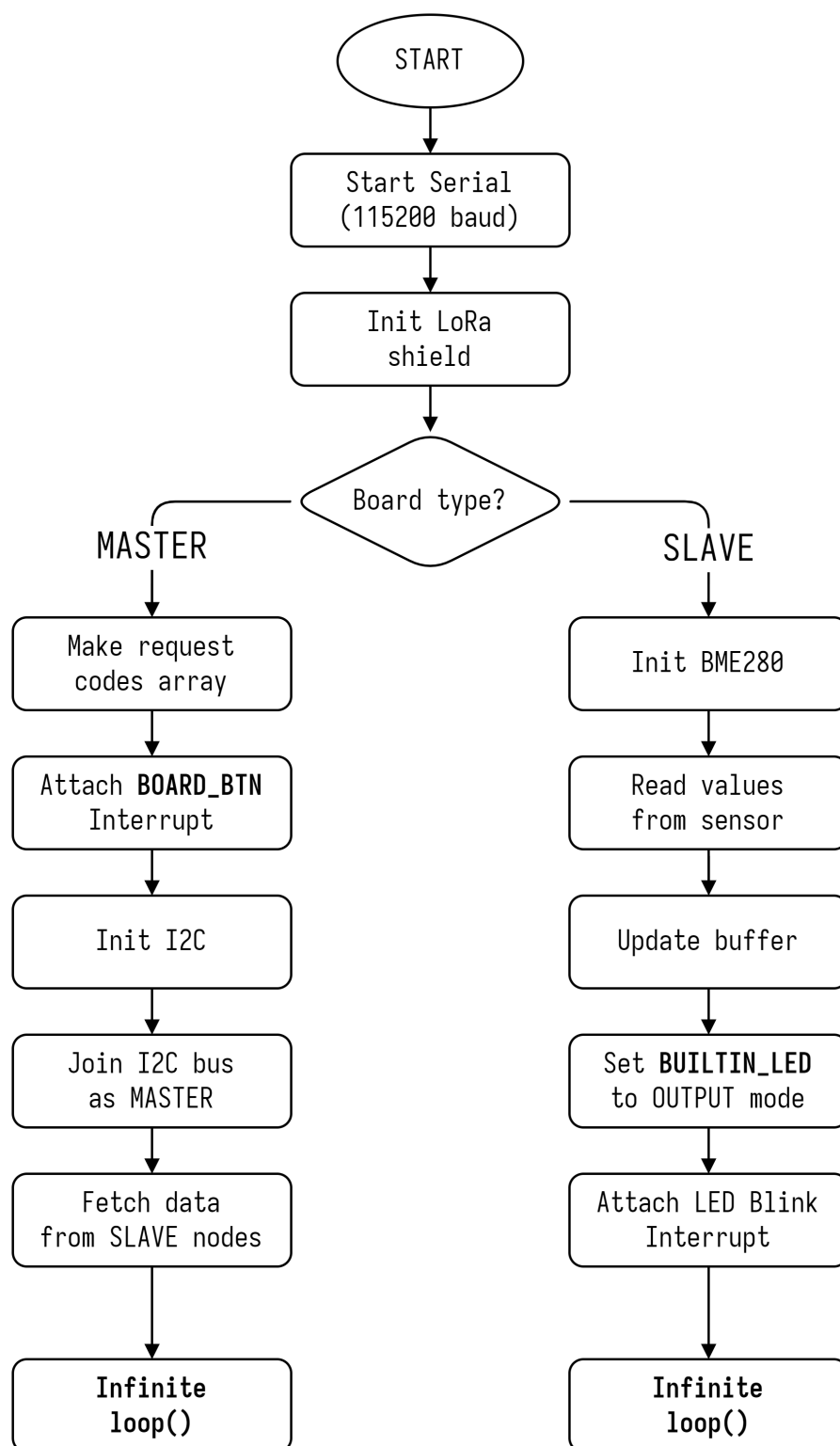
1. przygotowanie sensora BME280 oraz pobranie z niego danych,
2. aktualizacja zawartości bufora (wykorzystywanego do przechowywania odczytanych wartości),
3. przygotowanie diody LED, która informuje o trwającej komunikacji w sieci,
4. przygotowanie przerwania, wykorzystywanego do obsługi nowych zapytań.

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 ułatwiają 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



Rysunek 7. Schemat blokowy części `setup()` oprogramowania modułów sieci LoRa, z podziałem na typ płytki

zawierają informację o tym jaki kształt powinny mieć dane zbierane z sensorów oraz przekazywane 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()`.

```
1 #define DATAID_MASK(req) (req & 0xf0)
2 #define BOARDID_MASK(req) (req & 0x0f)
3 #define ARRAYSIZE(x) (sizeof(x) / sizeof(x[0]))
4
5 /**
6  * @brief Template struct for single value sensor data.
7  * @tparam T Field variable type
8  */
9 template <typename T>
10 struct SensorValues
11 {
12     T temperature;
13     T pressure;
14     T humidity;
15 };
16
17 /**
18  * @brief Template struct for sensor data that needs an array.
19  * @tparam T Field variable type
20  * @tparam N Array size
21  */
22 template <typename T, size_t size>
23 struct SensorData
24 {
25     T temperature[size];
26     T pressure[size];
27     T humidity[size];
28 };
```

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


```

1  /**
2   * @brief Prints out a debug message line over Serial
3   *
4   * @param type: Debug message type @ref MsgType_t
5   * @param msg: Message string
6   */
7  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. 8 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.

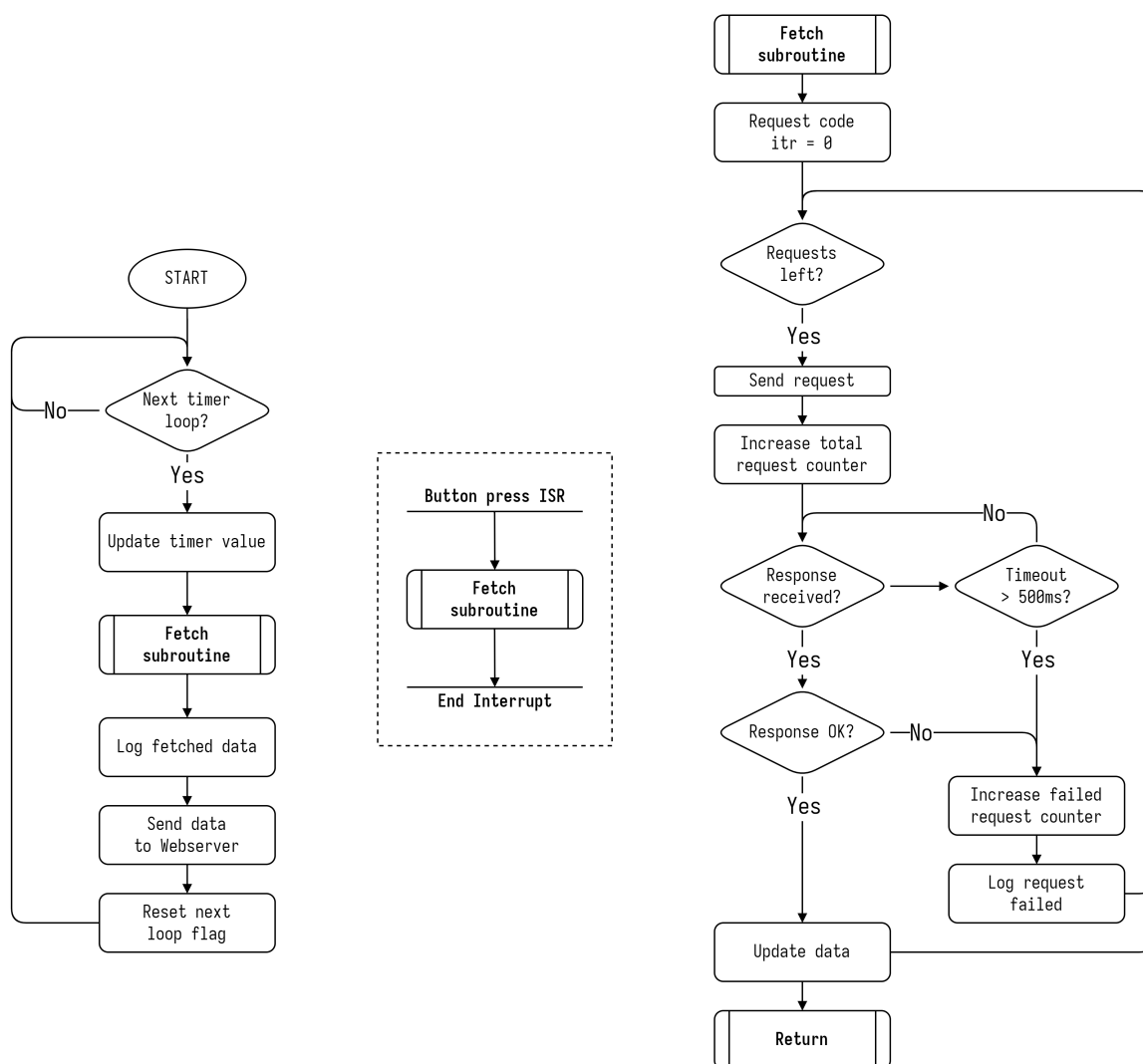
```

1  case lora::MASTER:
2      if (next)
3      {
4          timer = millis();
5          fetchSubroutineHandler();
6          INVERT(next);
7      }
8      break;
9
10 if ((millis() - timer) >= PERIOD_MS)
11 {
12     INVERT(next);
13 }

```

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

```
1 void fetchSubroutineHandler(void)
2 {
3     for (u8 code : requestCode)
4     {
5         fetchData(code);
6     }
7
8     logReceivedData(&receivedData);
9     webserverTransmit(&receivedData);
10 }
11
12 void fetchData(u8 requestCode)
13 {
14     lora::sendRequest(requestCode);
15     totalRequests[(BOARDID_MASK(requestCode)) - 1] += 1;
16
17     // If MASTER waits 500ms for response, treat fetch as failed
18     u32 timeout = millis();
19     while (!(loraRadio.read(receivedMsg) > 0))
20     {
21         if ((millis() - timeout) >= TIMEOUT_MS)
22         {
23             timeout = millis();
24             debug::println(debug::ERR, "Request 0x" + String(requestCode,
25                                     HEX) +
26                                     " failed: TIME OUT.");
27             failedRequests[(BOARDID_MASK(requestCode)) - 1] += 1;
28             return;
29         }
30
31         // No timeout, check response code matches request code
32         if (receivedMsg[0] != requestCode)
33         {
34             debug::println(debug::ERR, "Request 0x" + String(requestCode, HEX)
35                             +
36                             " failed: BAD RESPONSE");
37             failedRequests[(BOARDID_MASK(requestCode)) - 1] += 1;
38             return;
39         }
40
41         lora::readResponse(&receivedData, receivedMsg);
42         delay(100); // 100ms blocking delay between requests
43     }
```

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 request with value 0x" +
5         String(message, HEX));
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 zostanie 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
9             [2]) / 100;
10        break;
11
12    case PRESSURE:
13        data->pressure[boardId] = (f32)((message[1] << 8) + message[2])
14            ;
15        break;
16
17    case HUMIDITY:
18        data->humidity[boardId] = (f32)((message[1] << 8) + message[2])
19            / 100;
20        break;
21    }
22 }
```

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 kilkakrotnie 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 wyznaczenia 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.

```
1 template <typename T, size_t size>
2 extern void logValues(const T (&array)[size])
3 {
4     for (T item : array)
5     {
6         Serial.print(item);
7         Serial.print("\t");
8     }
9     Serial.println();
10 }
```

Listing 8. Implementacja funkcji szablonowej `logValues()`

```
1 template <typename T, size_t size>
2 void transmitPacket(const T (&array)[size], u8 typeKey, f32 modifier)
3 {
4     char packet[50];
5     for (T item : array)
6     {
7         sprintf(packet, "%i:%i&", typeKey, (u16)(item * modifier));
8         Wire.write(packet);
9     }
10 }
```

Listing 9. Implementacja funkcji szablonowej transmitPacket()

```
1 void logReceivedData(lora::ReceivedData *data)
2 {
3     Serial.println();
4     debug::println(debug::INFO, "Fetched data:");
5
6     Serial.print("Temperature:\t");
7     logValues(data->temperature);
8     Serial.print("Pressure:\t");
9     logValues(data->pressure);
10    Serial.print("Humidity:\t");
11    logValues(data->humidity);
12    Serial.println();
13
14    debug::println(debug::INFO, "Requests statistics:");
15
16    Serial.print("Total:\t\t");
17    logValues(totalRequests);
18    Serial.print("Failed:\t\t");
19    logValues(failedRequests);
20    getFailedPercent(totalRequests, failedRequests, failedPercent);
21    Serial.print("Failed%:\t");
22    logValues(failedPercent);
23
24    Serial.println();
25 }
```

Listing 10. Funkcja wykorzystywana do logowania wartości przez port szeregowy

```
1 void webserverTransmit(lora::ReceivedData *data)
2 {
3     debug::println(debug::INFO, "Sending to webserver");
4     Wire.beginTransmission(I2C_ADDR);
5
6     transmitPacket(data->temperature, TEMPERATURE, 100.0f);
7     Serial.println("Temperature");
8     transmitPacket(data->pressure, PRESSURE);
9     Serial.println("Pressure");
10    transmitPacket(data->humidity, HUMIDITY, 100.0f);
11    Serial.println("Humidity");
12    getFailedPercent(totalRequests, failedRequests, failedPercent);
13    transmitPacket(failedPercent, FAILPERCENT, 100.0f);
14    Serial.println("Failed%");
15
16    Wire.endTransmission();
17 }
```

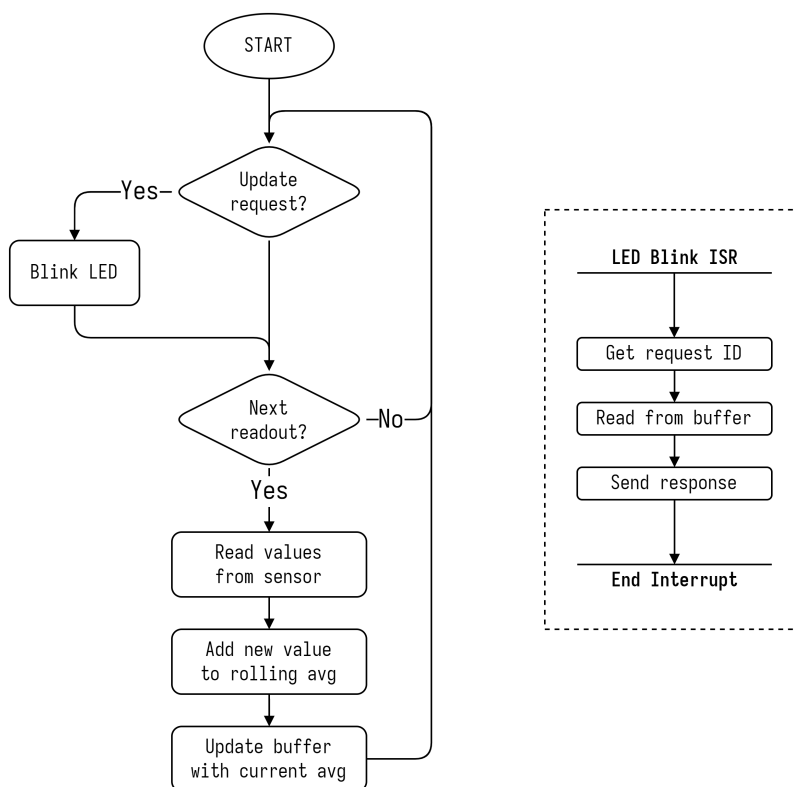
Listing 11. Funkcja do transmisji danych do modułu serwera przez magistralę I2C

```
1 void getFailedPercent(u8 (&totalReq)[3], u8 (&failReq)[3],
2                       f32 (&failPercent)[3])
3 {
4     for (u8 i = 0; i < sizeof(totalReq); ++i)
5     {
6         failedPercent[i] = ((f32)failReq[i] / (f32)totalReq[i]) * 100.0f;
7     }
8 }
```

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. 9 przedstawiony został diagram nieskończonej pętli zaimplementowanej dla modułów SLAVE.



Rysunek 9. 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 aktualizacja 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


```
1  case lora::SLAVE:
2      if (loraRadio.read(updateRequestMsg))
3      {
4          // This will trigger an interrupt
5          digitalWrite(LED_BUILTIN, 1);
6      }
7
8      if (next)
9      {
10         sensor::readRaw();
11         sensor::updateBuffer(&sensorBuffer);
12         timer = millis();
13         INVERT(next);
14     }
15     break;
16
17     if ((millis() - timer) >= PERIOD_MS)
18     {
19         INVERT(next);
20     }
```

Listing 13. Implementacja nieskończonej pętli dla modułów SLAVE

```
1  void readRaw(void)
2  {
3      temperatureAvg.addValue(bme.readTemperature());
4      pressureAvg.addValue(bme.readPressure());
5      humidityAvg.addValue(bme.readHumidity());
6  }
```

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

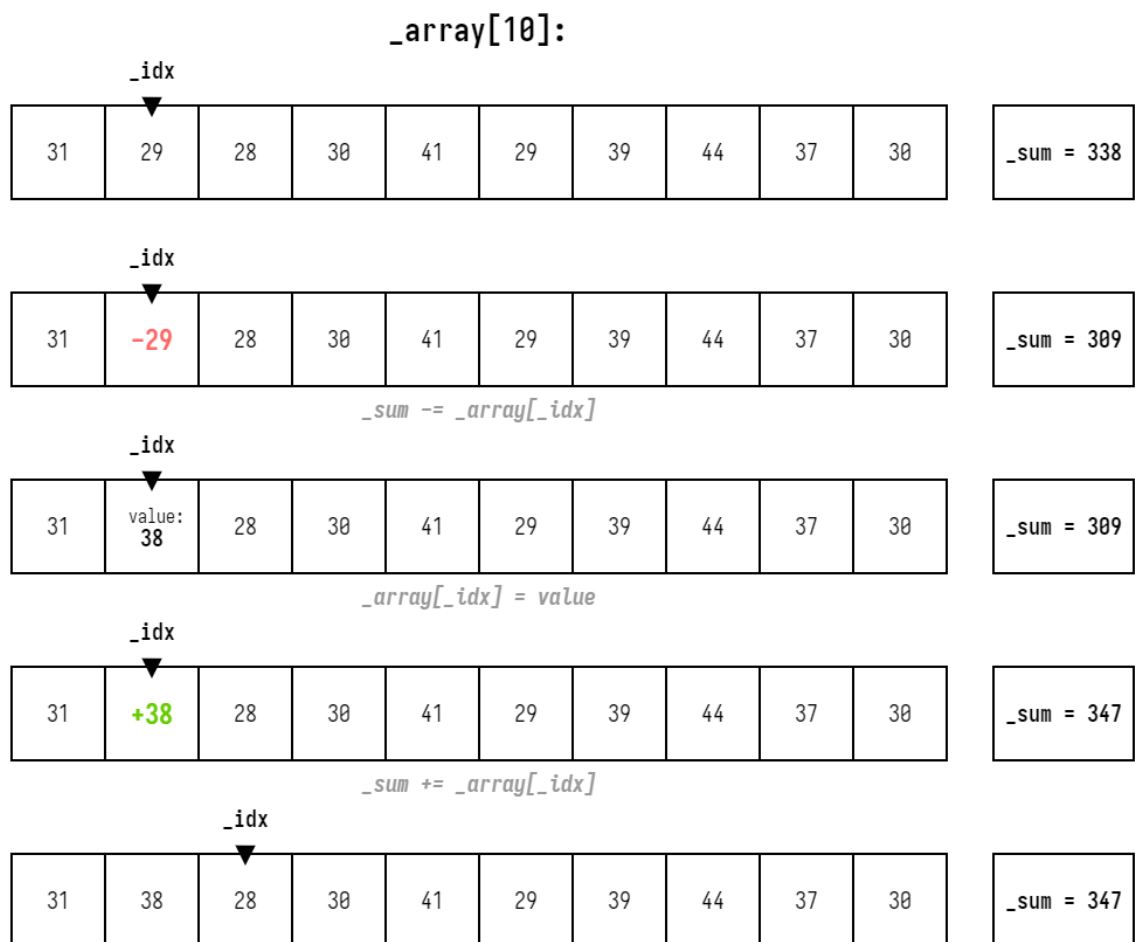
optymalnego działania kodu – średnia krocząca jest zadaniem dość intensywnym obliczeniowo – zastosowane zostało przydzielanie pamięci (ang. *memory allocation*) poprzez `malloc()` oraz wypełnianie tablicy samymi zerami przy początkowej deklaracji obiektu, dzięki czemu dodawanie wartości jest znacznie 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. 10, natomiast na listingu 16 jego implementacja.

```
1 class RollingAvg
2 {
3 public:
4     RollingAvg(u8 nSample);
5     ~RollingAvg();
6
7     void clear(void);
8     void addValue(f32 value);
9     f32 getAverage(void);
10
11 private:
12     u8 _size;
13     u8 _count;
14     u8 _idx;
15     float _sum;
16     float *_array;
17 };
```

Listing 15. Definicja klasy `RollingAvg`

```
1 void RollingAvg::addValue(f32 value)
2 {
3     if (_array == NULL)
4         return;
5
6     _sum -= _array[_idx];
7     _array[_idx] = value;
8     _sum += _array[_idx];
9     ++_idx;
10
11     if (_idx == _size) // Set _idx to 0 if max reached
12         _idx = 0;
13
14     if (_count < _size)
15         ++_count;
16 }
```

Listing 16. Implementacja algorytmu dodającego nową wartość do średniej kroczącej



Rysunek 10. 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.

```
1 void updateBuffer(BufferData *buffer)
2 {
3     buffer->temperature = (u16)(temperatureAvg.getAverage() * 100.0f);
4     buffer->pressure = (u16)(pressureAvg.getAverage() / 100.0f);
5     buffer->humidity = (u16)(humidityAvg.getAverage() * 100.0f);
6 }
7 }
```

Listing 17. Implementacja funkcji aktualizującej zawartość bufora na dane

```
1 float RollingAvg::getAverage()
2 {
3     if (_count == 0)
4         return NAN;
5
6     return (_sum / _count);
7 }
```

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.

```

1 void newRequestHandler(void)
2 {
3     if (BOARDID_MASK(updateRequestMsg[0]) != BOARD_ID)
4     {
5         digitalWrite(LED_BUILTIN, 0);
6         return;
7     }
8
9     lora::sendResponse(&sensorBuffer, updateRequestMsg[0]);
10    digitalWrite(LED_BUILTIN, 0); // Turn off the LED when done, visual
    indicator
11 }

```

Listing 19. Implementacja funkcji obsługującej przerwania w modułach SLAVE

```

1 void sendResponse(sensor::BufferData *buffer, u8 reqMsg)
2 {
3     /**
4      * @brief Payload of the response. First byte is request code echo,
5      * then 2 bytes of response, MSB-order.
6      */
7     u8 message[3];
8     u16 bufferValue;
9
10    switch (DATAID_MASK(reqMsg))
11    {
12    case TEMPERATURE:
13        bufferValue = (u16)buffer->temperature;
14        break;
15
16    case PRESSURE:
17        bufferValue = (u16)buffer->pressure;
18        break;
19
20    case HUMIDITY:
21        bufferValue = (u16)buffer->humidity;
22        break;
23    }
24
25    message[0] = reqMsg;
26    message[1] = (bufferValue & UPPER_BITMASK) >> 8;
27    message[2] = (bufferValue & LOWER_BITMASK);
28    debug::println(debug::INFO, "Sending response: 0x" + String(message
29        [0], HEX) + "\t" + String(message[1]) + "\t" + String(message
30        [2]));
31
32    loraRadio.write(message, sizeof(message));
33 }

```

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 przysyłać 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.

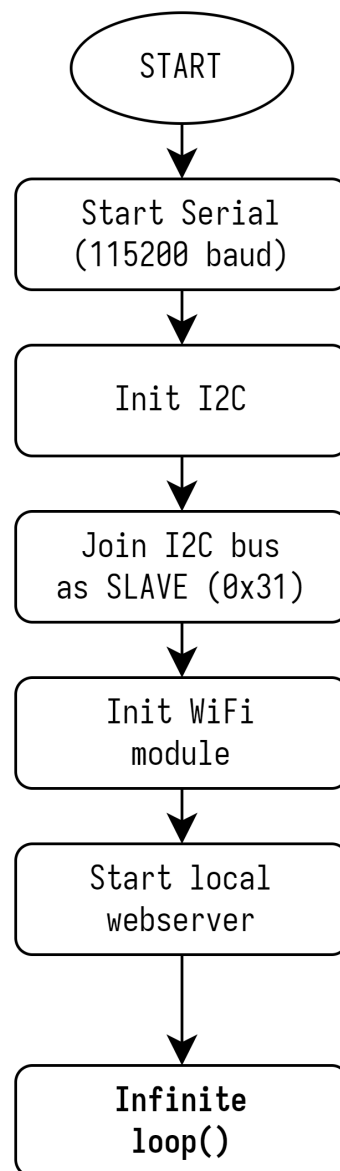
4.3 Implementacja oprogramowania modułu serwera sieciowego

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. 11, natomiast na listingu 21 przedstawiona została implementacja tej części oprogramowania.

```
1 void setup()
2 {
3     Wire.begin(I2C_ADDR);
4     Wire.onReceive(i2cReceive);
5     Serial.begin(115200);
6
7     wifi::init();
8     webserver::init();
9 }
```

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 11. Diagram blokowy sekcji setup() oprogramowania modułu serwera sieciowego

```
1  void init(void)
2  {
3      debug::println(debug::INFO, "Trying to init WiFi module...");
4      WiFi.setPins(WIFISHIELD_PINS);
5
6      // Check if WiFi module is present
7      if (WiFi.status() == WL_NO_SHIELD)
8      {
9          debug::println(debug::ERR, "WiFi module init fail (disconnected
10             ?)");
11          while (true)
12          {
13              // Pause forever
14          }
15
16          pinMode(LED_BUILTIN, OUTPUT);
17          debug::println(debug::INFO, "Attempting connection to: " + String(
18             WIFI_SSID));
19
20          while (WiFi.status() != WL_CONNECTED)
21          {
22              WiFi.begin(WIFI_SSID, WIFI_PASS);
23
24              digitalWrite(LED_BUILTIN, 1);
25              delay(250);
26              digitalWrite(LED_BUILTIN, 0);
27              delay(250);
28          }
29
30          digitalWrite(LED_BUILTIN, 1);
31          printStatus();
32      }
```

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

```
1  WiFiServer server(SERVER_PORT);
2  void init(void)
3  {
4      while (WiFi.status() != WL_CONNECTED)
5      {
6          // Wait for WiFi to connect before starting server
7      }
8
9      debug::println(debug::INFO, "Webserver started on port " + String(
10         SERVER_PORT, DEC));
11      server.begin();
12  }
```

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

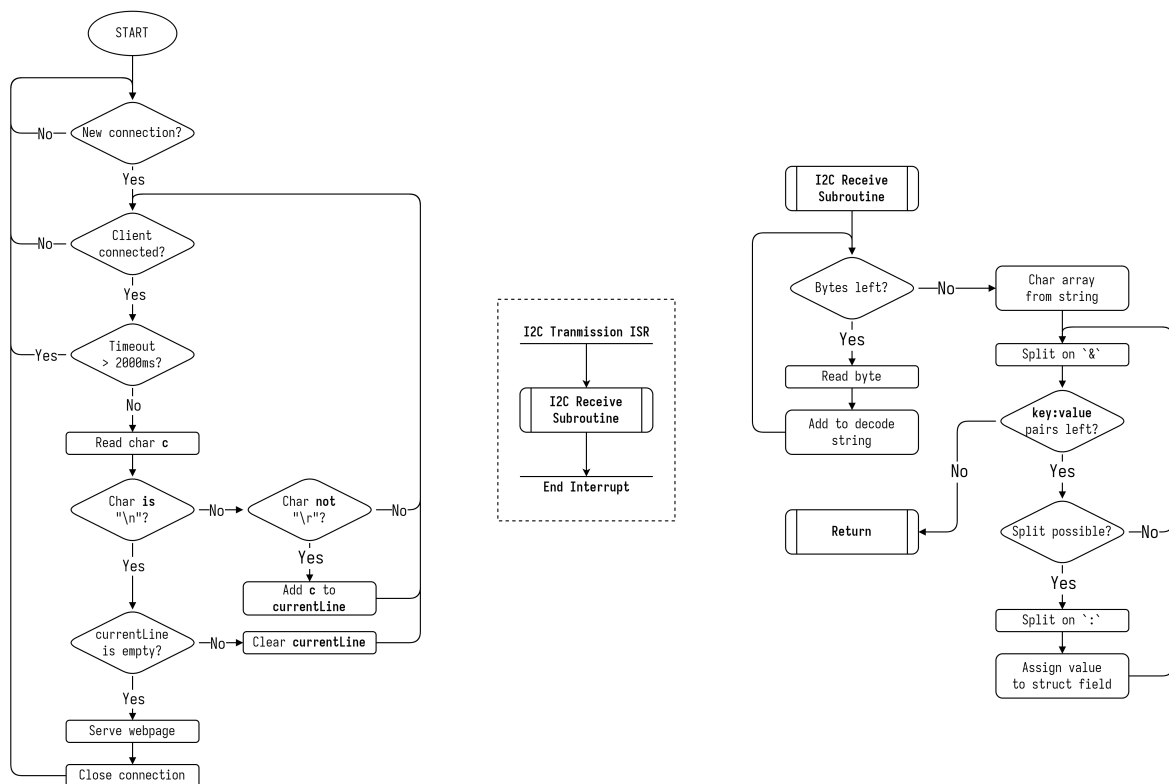
Inicjalizacja WiFi wykorzystuje zdefiniowane w osobnym pliku konfiguracyjnym SSID (nazwę) oraz hasło do sieci WiFi, gdzie moduł ma wykonać próbę podłączenia. Do implementacji funkcji została dodana także funkcja sprawdzenia czy rozszerzenie WiFi na płytce Feather jest poprawnie podłączone. Gdy moduł 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. 12 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 12. 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. 13 przedstawiony został schemat blokowy działania obu tych funkcji.



Rysunek 13. Schemat bloków działania funkcji w nieskończonej pętli serwera sieciowego

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 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.

```
1      char c = wifi::client.read();
2      webserver::header += c;
3
4      if (c == '\n')
5      {
6          if (currentLine.length() == 0)
7          {
8              webserver::serve(&storedData);
9              break;
10         }
11         currentLine = "";
12     }
13     else if (c != '\r')
14     {
15         currentLine += c;
16     }
```

Listing 25. Implementacja odbierania przychodzącego żądania

Przesyłanie odpowiedzi przez serwer wykonywane wykorzystując do tego zaimplementowaną funkcję `webserver::serve()`. Jej implementacja przedstawiona 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, nagłówki 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 wykorzystując 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.

```

1  void serve(StoredData_t *data)
2  {
3      for (String line : httpHeader)
4      {
5          wifi::client.println(line);
6      }
7      wifi::client.println();
8
9      for (String line : webpageHead)
10     {
11         wifi::client.println(line);
12     }
13
14     for (u8 i = 0; i < ARRAYSIZE(webpageBody); ++i)
15     {
16         if (i == BODY_TAB_IDX)
17         {
18             printRow(data->temperature, "Temperature [&deg;C]");
19             printRow(data->pressure, "Pressure [hPa]");
20             printRow(data->humidity, "Humidity [%]");
21             printRow(data->failedPercent, "Failed req. [%]");
22             continue; // Skip printing from array
23         }
24         wifi::client.println(webpageBody[i]);
25     }
26 }

```

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

```

1  template <typename T, size_t size, size_t length>
2  void printRow(const T (&array)[size], const char (&rowTitleStr)[length])
3  {
4      static u8 fmtCharCount = 15;
5      char line[length + fmtCharCount];
6
7      sprintf(line, "<tr><td>%s</td>", rowTitleStr);
8      wifi::client.println(line);
9
10     // Template type does not matter, values cast to f32 for display
11     for (T value : array)
12     {
13         char cell[25];
14         sprintf(cell, "<td>%.1f</td>", (f32)value);
15         wifi::client.println(cell);
16     }
17
18     wifi::client.println("</tr>");
19 }

```

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 otrzymywanych 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.

```
1 void i2cReceive(int byteCount)
2 {
3     // Build a string from I2C transmission
4     String recv;
5     while (Wire.available() > 0)
6     {
7         char recvChar = Wire.read();
8         recv += recvChar;
9     }
10
11     // Decode into values
12     decodeStr(&recv, &storedData);
13 }
```

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 uzupełnianej 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 algorytm dotrze do końca tablicy.

```

1 void decodeStr(String *recv, StoredData_t *data)
2 {
3     char recvArray[recv->length() + 1];
4     recv->toCharArray(recvArray, recv->length());
5     recvArray[recv->length()] = 0;
6
7     // Keeps count for each struct member
8     u8 dataCounter[4] = {0, 0, 0, 0};
9
10    /**
11     * Read key-value pairs
12     * Split string on `&` -> each key-value pair
13     * Split pairs on `:` -> key : value
14     */
15    char *keyToken = strtok(recvArray, "&");
16    while (keyToken != NULL)
17    {
18        char *valueToken = strchr(keyToken, ':');
19        if (valueToken != 0)
20        {
21            *valueToken = 0;
22            int key = atoi(keyToken);
23            ++valueToken;
24
25            // Assign value from valueToken to correct struct member
26            switch (key)
27            {
28                case TEMPERATURE:
29                    data->temperature[dataCounter[TEMPERATURE]++] =
30                        (f32)atoi(valueToken) / 100.0f;
31                    break;
32                case PRESSURE:
33                    data->pressure[dataCounter[PRESSURE]++] = (f32)atoi(
34                        valueToken);
35                    break;
36                case HUMIDITY:
37                    data->humidity[dataCounter[HUMIDITY]++] =
38                        (f32)atoi(valueToken) / 100.0f;
39                    break;
40                case FAILPERCENT:
41                    data->failedPercent[dataCounter[FAILPERCENT]++] =
42                        (f32)atoi(valueToken) / 100.0f;
43                    break;
44            }
45            keyToken = strtok(NULL, "&");
46        }
47    }

```

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

Rozdział 5

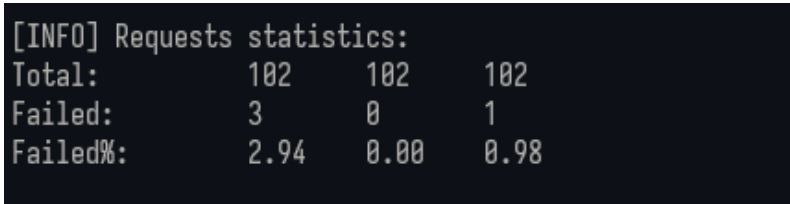
Testy implementacji

W celu zweryfikowania czy zaimplementowane oprogramowanie działa poprawnie wykonano zestaw testów. Sprawdzone zostało czy moduły komunikują się poprawnie między sobą (między modułami sieci oraz modułem MASTER a modułem serwera sieciowego) oraz czy dane, które moduł serwera sieciowego odebrał wyświetlane są poprawnie na zaimplementowanej stronie internetowej.

5.1 Testy komunikacji sieci

Do sprawdzenia poprawności komunikacji wewnątrz sieci wykorzystana została implementacja funkcji, która zlicza ilość wysłanych zapytań przez moduł MASTER oraz ilość otrzymanych odpowiedzi oraz funkcja, która na podstawie otrzymanych wartości oblicza wartość procentową zapytań z błędem (niezależnie od tego czy był to timeout – przekroczenie czasu oczekiwania na odpowiedź, czy otrzymana odpowiedź nie zgadzała się z tą, na jaką moduł MASTER oczekiwał). Kod źródłowy odpowiedzialnych za to elementów przedstawiony został na listingach 30 oraz 31.

Sieć została uruchomiona na pewien czas i podczas jej pracy monitorowane było czy komunikacja działa poprawnie. Podczas testu wykorzystano jedynie podgląd sieci poprzez monitor portu szeregowego na moduł MASTER. Na rys. 14 przedstawiony został zrzut ekranu pokazujący, że sieć komunikuje się poprawnie oraz to, że oprogramowanie zlicza całkowitą ilość zapytań oraz ilość błędów w komunikacji wraz z procentowym błędem.



```
[INFO] Requests statistics:
Total:      102    102    102
Failed:      3      0      1
Failed%:     2.94   0.00   0.98
```

Rysunek 14. Zrzut ekranu z monitora portu szeregowego podczas testowania komunikacji w sieci.

```
1  totalRequests[(BOARDID_MASK(requestCode)) - 1] += 1;
2
3  // If MASTER waits 500ms for response, treat fetch as failed
4  if ((millis() - timeout) >= TIMEOUT_MS)
5  {
6      timeout = millis();
7      debug::println(debug::ERR, "Request 0x" + String(requestCode,
8          HEX) +
9          " failed: TIME OUT.");
10     failedRequests[(BOARDID_MASK(requestCode)) - 1] += 1;
11     return;
12 }
13
14 // No timeout, check response code matches request code
15 if (receivedMsg[0] != requestCode)
16 {
17     debug::println(debug::ERR, "Request 0x" + String(requestCode, HEX)
18         +
19         " failed: BAD RESPONSE");
20     failedRequests[(BOARDID_MASK(requestCode)) - 1] += 1;
21     return;
22 }
```

Listing 30. Zliczanie całkowitej ilości zapytań oraz odpowiedzi zakończonych błędem (timeout, zły nagłówek odpowiedzi)

```
1 void getFailedPercent(u8 (&totalReq)[3], u8 (&failReq)[3],
2                       f32 (&failPercent)[3])
3 {
4     for (u8 i = 0; i < sizeof(totalReq); ++i)
5     {
6         failedPercent[i] = ((f32)failReq[i] / (f32)totalReq[i]) * 100.0f;
7     }
8 }
```

Listing 31. Implementacja funkcji pomocniczej do wyznaczania wartości procentowej zapytań z błędem

5.2 Testy transmisji danych do serwera sieciowego

Wykonane zostały także testy komunikacji modułu MASTER z modułem serwera sieciowego. Sprawdzone zostało czy dane zbierane przez sieć LoRa poprawnie przesyłane przez magistralę I2C oraz dekodowane po stronie modułu serwera sieciowego.

Korzystając z zaimplementowanego logowania przez port szeregowy modułu MASTER informacji o transmisji przez magistralę I2C (listing 11) oraz dodatkowej funkcjonalności w module serwera sieciowego, której implementacja przedstawiona została na listingu 32, logującej przez port szeregowy ilość odebranych transmisji, sprawdzona została poprawność komunikacji.

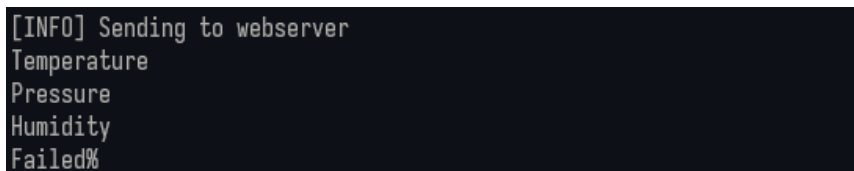
```

1  debug::println(debug::INFO, "Transmission counts");
2  for (u8 count : dataCounter)
3  {
4      Serial.print(count);
5      Serial.print(" ");
6  }
7  Serial.println();

```

Listing 32. Implementacja funkcjonalności zliczania transmisji z modułu MASTER na module serwera sieciowego.

Moduły zostały uruchomione oraz włączony został monitor portu szeregowego modułu MASTER oraz modułu serwera sieciowego. Przez pewien czas prowadzona była obserwacja logowanych wartości. Podczas trwania testu stwierdzone zostało, że moduły działają i komunikują się poprawnie. Przykładowe logi, pokazujące poprawną komunikację przedstawione zostały na rys. 15 oraz 16.

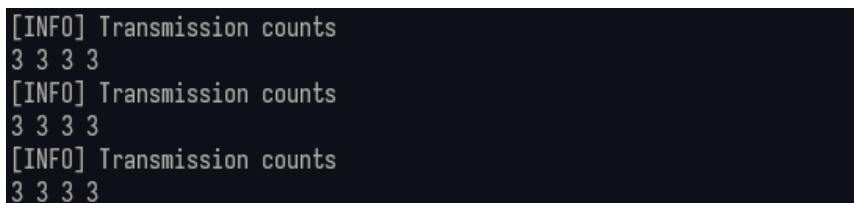


```

[INFO] Sending to webserver
Temperature
Pressure
Humidity
Failed%

```

Rysunek 15. Logi o transmisji przez magistralę I2C z modułu MASTER.



```

[INFO] Transmission counts
3 3 3 3
[INFO] Transmission counts
3 3 3 3
[INFO] Transmission counts
3 3 3 3

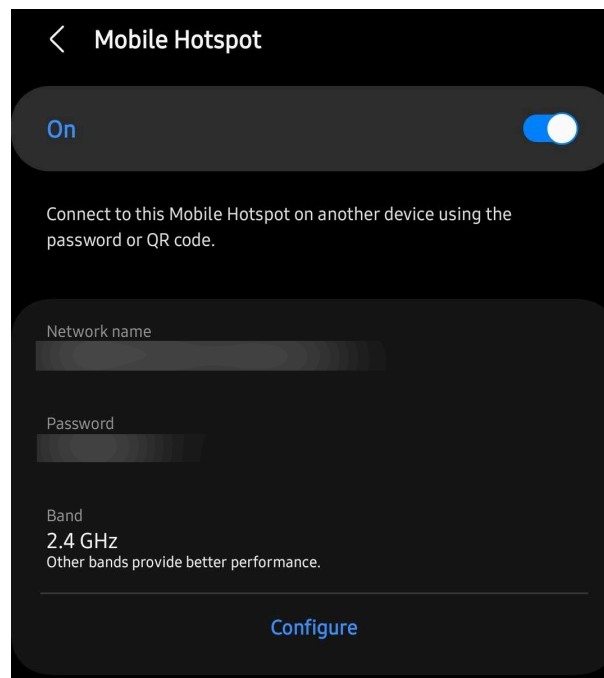
```

Rysunek 16. Logi pokazujące poprawne odbieranie wartości z magistrali I2C przez moduł serwera sieciowego.

5.3 Testy działania serwera sieciowego oraz strony internetowej

Ostatnim wykonanym testem, który pozwolił na stwierdzenie, że zaimplementowane oprogramowanie działa poprawnie było sprawdzenie samego serwera sieciowego. Zweryfikowane zostało czy uruchamia się on poprawnie, gdy pojawi się możliwość zalogowania do sieci lokalnej sieci WiFi (której dane zostały podane w pliku konfiguracyjnym) oraz czy poprawnie działa strona internetowa, którą moduł serwuje użytkownikowi, chcącemu zobaczyć zebrane przez sieć LoRa dane.

Do testu wykorzystany został mobilny hotspot (lokalna sieć WiFi utworzona przez smartphona) oraz zaimplementowana rejestracja informacji o logowaniu się modułu do sieci WiFi i uruchamiania serwera sieciowego. Na rys. 17 oraz 18 pokazana została włączony mobilny hotspot oraz logi z modułu serwera sieciowego. Na ich podstawie stwierdzone zostało, że serwer sieciowy działa poprawnie, ponieważ jednym z logów jest informacja o starcie serwera z jego lokalnym IP (umożliwiającym wyświetlenie strony internetowej) na porcie 80.



Rysunek 17. Zrzut ekranu pokazujący włączony mobilny hotspot.

```
[INFO] Connected to: [redacted] (-38dBm)
[INFO] Local IP: 192.168.137.101
[INFO] Webserver started on port 80
```

Rysunek 18. Logi pokazujące poprawne połączenie z siecią WiFi oraz uruchomienie serwera sieciowego.

Działanie zaimplementowanej strony internetowej sprawdzone zostało poprzez wykonanie próby załadowania jej w dwóch przypadkach: przed pierwszym przesłaniem danych z sieci LoRa (lub tuż po restarcie modułu serwera sieciowego) oraz po tym jak dane zostały przesłane. Miało to

na celu sprawdzenie, poza samym ładowaniem strony, także czy jej zawartość zostanie poprawnie zaktualizowana, gdy z modułu MASTER przesłane zostaną dane. Rys. 19 pokazuje poprawnie załadowaną stronę przed przesłaniem danych z sieci LoRa, natomiast rys. 20 to zrzut ekranu z widoczną poprawną aktualizacją danych po odebraniu ich przez moduł serwera sieciowego.

LoRa Private sensor network

MEASUREMENT	NODE 1	NODE 2	NODE 3
Temperature [°C]	0.0	0.0	0.0
Pressure [hPa]	0.0	0.0	0.0
Humidity [%]	0.0	0.0	0.0
Failed req. [%]	0.0	0.0	0.0

Rysunek 19. Zrzut ekranu ze strony bez danych z sieci.

LoRa Private sensor network

MEASUREMENT	NODE 1	NODE 2	NODE 3
Temperature [°C]	27.1	27.2	26.9
Pressure [hPa]	999.0	1000.0	999.0
Humidity [%]	36.1	35.4	36.2
Failed req. [%]	0.0	0.0	0.0

Rysunek 20. Zrzut ekranu ze strony po otrzymaniu danych przez moduł serwera sieciowego.

Rozdział 6

Badania działającej sieci

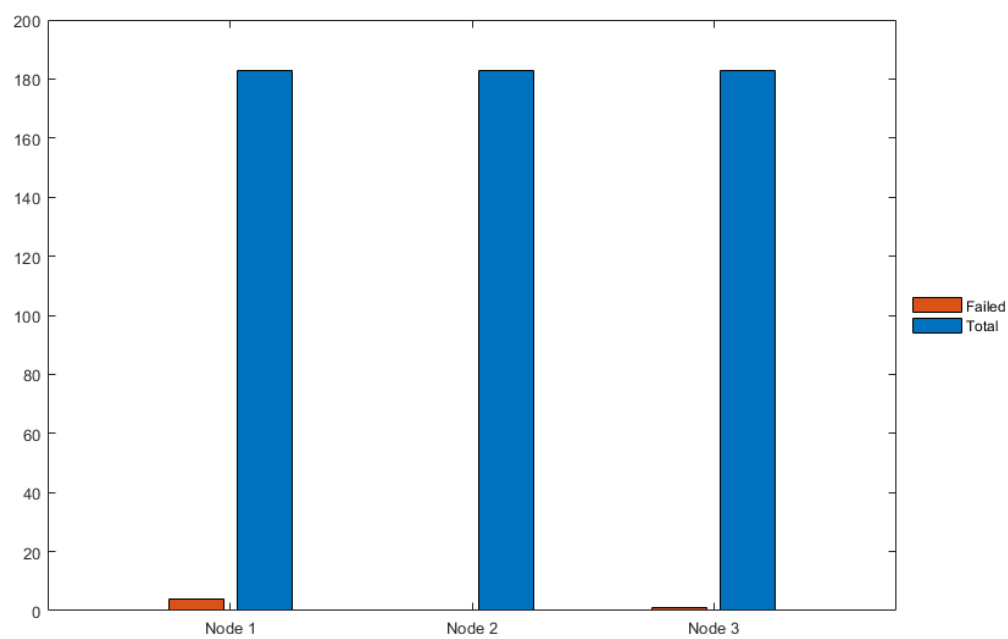
6.1 Badanie komunikacji w sieci

Podstawowym badaniem było było sprawdzenie poprawności komunikacji. Aby otrzymać zestaw danych testowych, moduły pozostały włączone, bez restartowania ich, przez 1 godzinę. Przy ustawieniu okresu pomiędzy wysyłaniem żądań przez moduł MASTER na 1 minutę dało to 61 pełnych wymian danych w sieci – 1 żądanie przy pierwszym uruchomieniu modułu MASTER oraz 60 podczas pracy modułu na zaimplementowanym zegarze (w sumie 183 żądania do każdego modułu SLAVE). Średnia odległość pomiędzy modułami podczas badania wynosiła 0.45m (45cm).

Tabela 1. Wyniki przeprowadzonego badania komunikacji w czasie 1 godziny.

Zapytania	SLAVE 1	SLAVE 2	SLAVE 3
W sumie [-]	183	183	183
Nieudane [-]	4	0	1
Nieudane [%]	2.19	0	0.55

Na podstawie danych sporządzony został wykres, przedstawiający otrzymane wyniki – sumę wysłanych zapytań oraz ilość, która zakończyła się błędem. Przedstawiony on został na rysunku 21.



Rysunek 21. Wykres słupkowy przedstawiający wyniki 1-godzinnego testu komunikacji 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

1	Schemat architektury sieci LoRaWAN	11
2	Tworzenie nowego projektu wykorzystując do tego PlatformIO Core (CLI)	14
3	Przykładowe wywołanie komendy <code>pio run</code> , tutaj do kompilacji projektu	16
4	Wyniki wyszukiwania bibliotek powiązanych z hasłem „LoRa”	17
5	Przykład wykorzystania komendy do instalacji bibliotek w projekcie	17
6	Schemat zbudowanej sieci, z oznaczonymi elementami komunikacji	21
7	Schemat blokowy części <code>setup()</code> oprogramowania modułów sieci LoRa, z podziałem na typ płytki	23
8	Schemat blokowy nieskończonej pętli oraz podprogramu zbierania danych zaimplementowanych dla modułu MASTER	26
9	Diagram blokowy pętli <code>loop()</code> zaimplementowanej dla modułów SLAVE	32
10	Schemat działania algorytmu dodawania nowej wartości do średniej kroczącej dla przykładowej tablicy 10 elementów, indeksu 2 oraz nowej wartości 38	35
11	Diagram blokowy sekcji <code>setup()</code> oprogramowania modułu serwera sieciowego	39
12	Wygląd strony internetowej zaimplementowanej do wizualizacji danych zbieranych z sieci	41
13	Schemat bloków działania funkcji w nieskończonej pętli serwera sieciowego	42
14	Zrzut ekranu z monitora portu szeregowego podczas testowania komunikacji w sieci.	49
15	Logi o transmisji przez magistralę I2C z modułu MASTER.	51
16	Logi pokazujące poprawne odbieranie wartości z magistrali I2C przez moduł serwera sieciowego.	51
17	Zrzut ekranu pokazujący włączony mobilny hotspot.	52
18	Logi pokazujące poprawne połączenie z siecią WiFi oraz uruchomienie serwera sieciowego.	52
19	Zrzut ekranu ze strony bez danych z sieci.	53
20	Zrzut ekranu ze strony po otrzymaniu danych przez moduł serwera sieciowego.	53
21	Wykres słupkowy przedstawiający wyniki 1-godzinnego testu komunikacji sieci	56