

*Pico Pagers*  
Urządzenia do powiadamiania klienta w restauracji  
Systemy Wbudowane

Jakub Kosmydel  
Norbert Morawski

16 czerwca 2023

## Spis treści

|          |                                   |           |
|----------|-----------------------------------|-----------|
| <b>1</b> | <b>Wprowadzenie</b>               | <b>2</b>  |
| <b>2</b> | <b>Uruchomienie</b>               | <b>2</b>  |
| 2.1      | Konfiguracja urządzenia . . . . . | 2         |
| <b>3</b> | <b>Protokół komunikacyjny</b>     | <b>3</b>  |
| 3.1      | Warstwa sprzętowa . . . . .       | 3         |
| 3.2      | Warstwa programowa . . . . .      | 4         |
| 3.2.1    | Implementacja . . . . .           | 7         |
| <b>4</b> | <b>System plików</b>              | <b>8</b>  |
| <b>5</b> | <b>WiFi</b>                       | <b>9</b>  |
| <b>6</b> | <b>Serwer HTTP</b>                | <b>9</b>  |
| <b>7</b> | <b>Kryptografia</b>               | <b>10</b> |

# 1 Wprowadzenie

Dummy

## 2 Uruchomienie

W celu skorzystania z systemu należy go początkowo skonfigurować. Do uruchomienia potrzebne są:

- sieć WiFi (2.4 GHz),
- urządzenie (np. telefon, komputer) połączone z tą siecią WiFi.

### 2.1 Konfiguracja urządzenia

Konfiguracja urządzenia przebiega w następujący sposób:

1. Podłączamy *Pagers Server* do zasilania.
2. Na dowolnym urządzeniu elektronicznym wyszukujemy sieć WiFi o nazwie *pagers-server*, łączymy się z nią podając hasło *password*.
3. Używając przeglądarki internetowej wchodzimy na adres <http://192.168.4.1>.
4. Wyszukujemy dostępne sieci WiFi używając przycisku *Initiate Scan*.

### Setup the WiFi connection

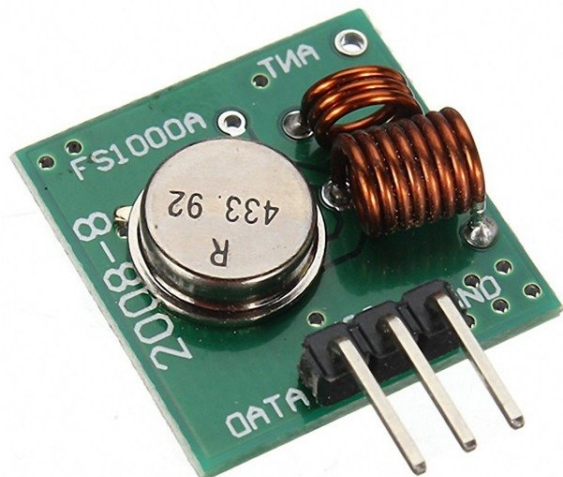
A green rectangular button with the text "Initiate Scan" in white.

Rysunek 1: Skanowanie sieci WiFi

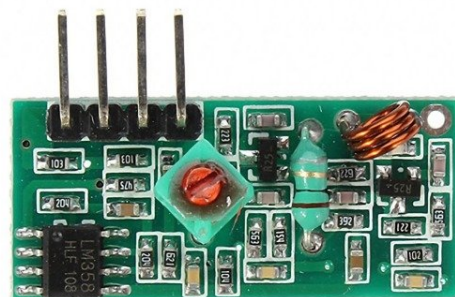
5. Wybieramy naszą sieć WiFi oraz podajemy do niej hasło. Klikamy *Connect* w celu połączenia się z wybraną siecią.

### 3 Protokół komunikacyjny

Zastosowaliśmy moduły komunikacyjne 433 MHz. Skłoniły nas do tego niska cena i prosta obsługa oraz brak wymaganej komunikacji zwrotnej przez nasze urządzenia.



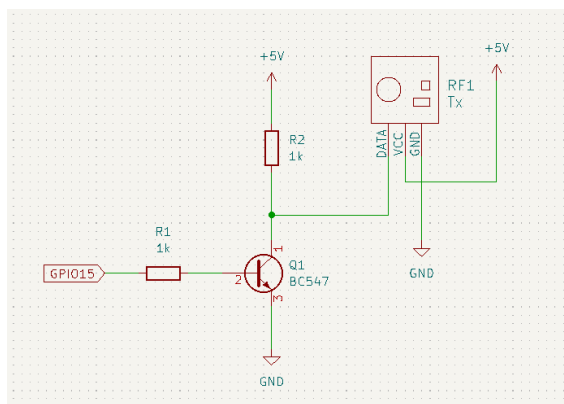
Rysunek 2: Nadajnik



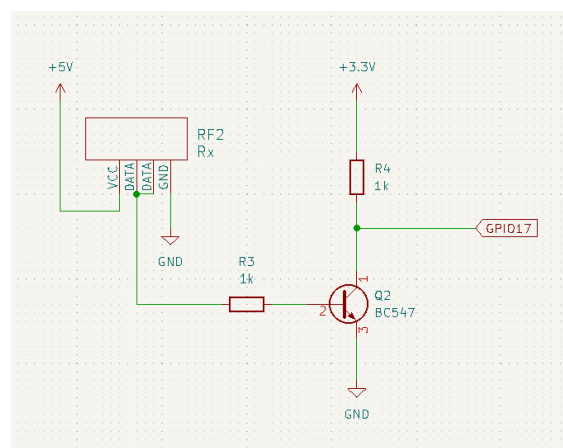
Rysunek 3: Odbiornik

#### 3.1 Warstwa sprzętowa

Obsługa nadajnika/odbiornika opiera się na podłączeniu zasilania i nadawania/odbierania poprzez jeden dostępny przewód danych. Niestety te moduły zasilane są napięciem 5V i taki standard napięcie stosują na wyjściu. Pi Pico toleruje tylko 3.3V. Konieczna więc była konwersja poziomów logicznych.



Rysunek 4: Układ nadajnika



Rysunek 5: Układ odbiornika

Podwójne odwracanie sygnału przez tranzystory niweluje się. Na wejściu odbiornika dostajemy nieodwrócony sygnał z nadajnika.

## 3.2 Warstwa programowa

Pierwszą naszą próbą było wykorzystanie wbudowanej komunikacji UART. Jednak okazało się że moduły te niezbyt dobrze przenoszą niezmienny się sygnał (co widać poniżej).



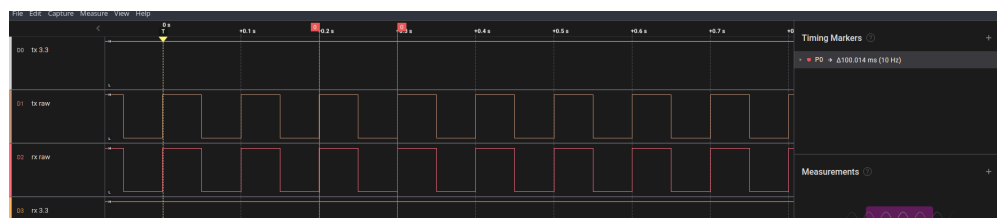
Rysunek 6: Wiadomość po stronie nadajnika/odbiornika

Po ok. 70 ms układ odbiornika zaczyna generować zakłócenia, które mogą być nieprawidłowo interpretowane jako sygnały komunikacji.



Rysunek 7: Wyodrębnione zjawisko niestabilności

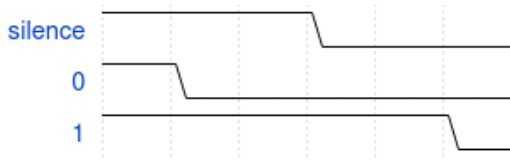
Po dokładnej analizie przebiegów odbiornika wynikło, że stan niski może być utrzymany przez ok. 160 ms, a stan wysoki przez około 72ms. Oznacza to że sygnał musi utrzymywać minimalną częstotliwość 14Hz.



Rysunek 8: Fala przenoszona bez zniekształceń

Przy ciągłych zmianach 10Hz okazuje się wystarczające.

Zaistniała potrzeba implementacji protokołu który utrzymywałby stałą częstotliwość fali nośnej. Przydatny w generacji takiego przebiegu jest PWM. Sterując wypełnieniem impulsu możemy przekazywać informacje binarne.



Rysunek 9: Protokół oparty o PWM

```
// Config
const int SUB_CYCLES = 6;
const int SUB_CYCLES_HIGH_SILENCE = 3;
// transmitter
const int SUB_CYCLES_HIGH_ZERO = 1;
const int SUB_CYCLES_HIGH_ONE = 5;
// receiver allowed
const int SUB_CYCLES_HIGH_ZERO_MAX = 2;
const int SUB_CYCLES_HIGH_ONE_MIN = 4;
```

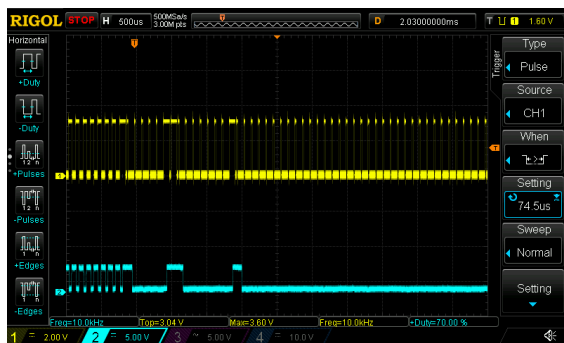
Rysunek 10: Konfiguracja protokołu

0.5 W obecnej wersji (konfigurowalne) zastosowaliśmy podziałkę  $\frac{1}{6}$  wypełnienia PWM.

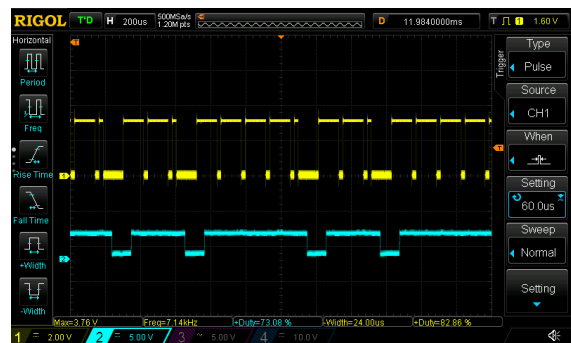
- Cisza to  $\frac{3}{6}$  wypełnienia,
- 0 to  $\frac{1}{6}$ ,
- 1 to  $\frac{5}{6}$ .

Odbiornik akceptuje 0 jako maksymalnie  $\frac{2}{6}$  wypełnienia, a 1 jako minimalnie  $\frac{4}{6}$  wypełnienia.

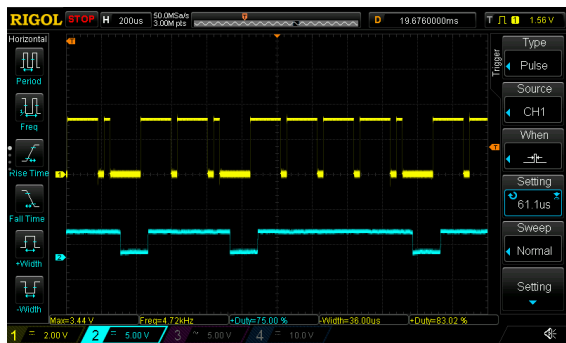
Producent określa maksymalną prędkość transmisji na 9600b/s (sugerowałoby to 9600Hz, jeżeli sygnalizowanie jest dwupoziomowe). Jednak generowane krótkie sygnały niekiedy są gubione przez nadajnik.



Rysunek 11: Gubienie impulsów 9600Hz



Rysunek 12: Gubienie impulsów 7200Hz



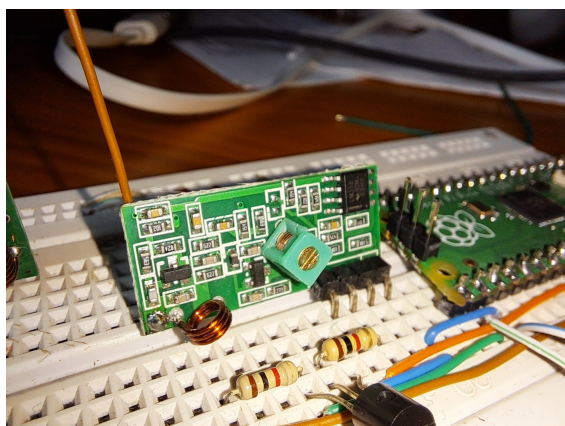
Rysunek 13: Gubienie impulsów 4800Hz



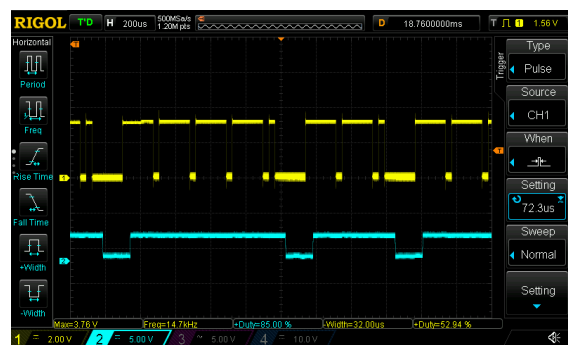
Rysunek 14: 2400Hz

Dopiero przy częstotliwości 2400Hz, wszystkie krótkie impulsy dotarły do odbiornika.

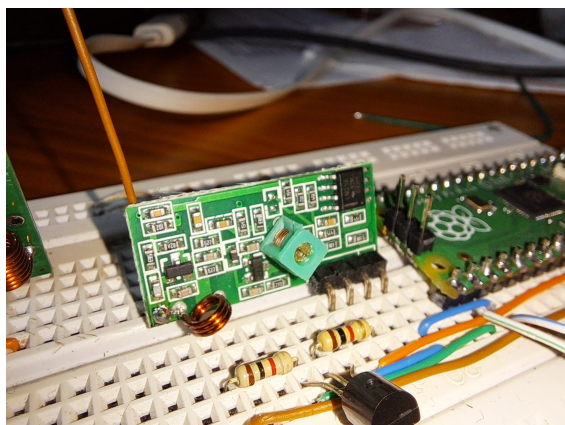
**Strojenie** Na płycie odbiornika dostępna jest cewka z możliwością dostrajania. Podjęliśmy próby jej nastawienia. Udało się osiągnąć szybkość transmisji 4800 b/s. Dla wyższych częstotliwości dostrajanie nie przyniosło efektów.



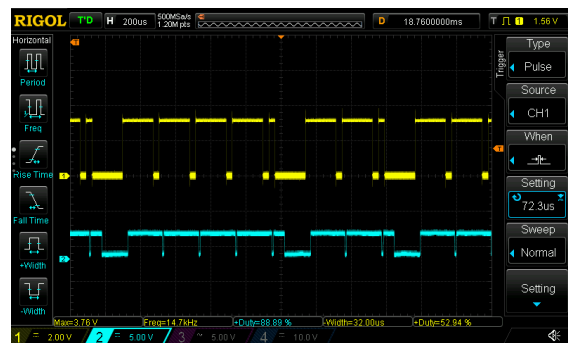
Rysunek 15: Cewka przed dostrajaniem



Rysunek 16: Przebieg 4800Hz przed dostrajaniem



Rysunek 17: Cewka po dostrojeniu



Rysunek 18: Przebieg 4800Hz po dostrojeniu

Zostaliśmy jednak przy transmisji 2400 b/s. Jest bardziej niezawodna, a szybkość nie ma dla nas wielkiego znaczenia. Nasza ramka danych ma rozmiar 16 bajtów. Przy 2400 b/s czas transmisji 1 ramki wynosi 53ms. Jest to bardzo mało w porównaniu do tego jak często będą wysyłane takie ramki.

### 3.2.1 Implementacja

Wysyłanie zostało zrealizowane z wykorzystaniem sprzętowego PWM i przerwania od jego przepełnienia. Częstotliwość PWM równa jest częstotliwości sygnalizowania w transmisji. Po wywołaniu przerwania przepełnienia, poziom wypełnienia ustawiany jest w zależności od następnego bitu danych. Jeżeli takiego nie ma, nadawana jest cisza.

```
void pwm_wrap_irq() {
    if (pwm_get_irq_status_mask() & (1 << slice_tx)) {
        pwm_clear_irq( slice_num: slice_tx);

        if (!tx_transfer) {
            return;
        }

        if (tx_bit_index == 8) {
            tx_bit_index = 0;
            tx_byte_index++;
            if (tx_byte_index == tx_byte_count) {
                tx_transfer = false;
                pwm_set_gpio_level( gpio: PIN_TX, level: PWM_DUTY_SILENCE);
                return;
            }
        }

        uint bit = (tx_bytes[tx_byte_index] << tx_bit_index) & 0x80;
        pwm_set_gpio_level( gpio: PIN_TX,
                           level: bit ? PWM_DUTY_ONE : PWM_DUTY_ZERO);
        tx_bit_index++;
    }
}
```

Rysunek 19: Nadawanie PWM

Odbieranie natomiast wykorzystuje funkcję PWM mikrokontrolera RP2040, która umożliwia uruchomienie licznika w zależności od stanu pinu (obsługiwane są tylko piny nieparzyste). Używane jest także przerwanie na tym samym pinie, które wykrywa zbocze opadające (początek bitu). Zeruje ono licznik PWM, i czeka na kolejne zbocze opadające. Przy kolejnym zboczu wartość licznika jest interpretowana.

```

void rx_fall_callback(uint gpio, uint32_t events) {
    gpio_acknowledge_irq(gpio, event_mask: events);
    uint cnt = pwm_get_counter( slice_num: slice_rx);
    pwm_set_counter( slice_num: slice_rx, c: 0);

    uint64_t now = time_us_64();

    int bit;
    if (cnt < PWM_DUTY_ZERO_MAX) {
        bit = 0;
    }
    else if (cnt > PWM_DUTY_ONE_MIN) {
        bit = 1;
    }
    else {
        // silence
        bit = -1;
        if (now - last_good_bit > SPACING_ALLOWED_US_MAX) {
            // end of frame
            if (rx_byte_index > 0)
                cb( buf: rx_bytes, bytes: rx_byte_index);

            rx_byte_index = 0;
            rx_bit_index = 0;
        }
    }
}

```

Rysunek 20: Odbieranie PWM

Koniec ramki jest sygnalizowany przerwą w transmisji (podobnie do protokołu MOD-BUS). 10 znaków przerwy oznacza koniec ramki, przy czym nadajnik generuje 20 znaków przerwy.

## 4 System plików

Do implementacji przechowywania plików (głównie statycznych plików strony WWW) został użyty system plików LittleFS. Przy użyciu funkcji dostępu do pamięci Flash, zapisuje on dane w dostępnej pamięci na płycie Pi Pico.



```

// Read a region in a block. Negative error codes are propagated
// to the user.
int pico_read(const struct lfs_config *c, lfs_block_t block, lfs_off_t off, void *buffer, lfs_size_t size) {
    memcpy(buffer,
           (const void*)(FS_BASE_ABS + block * FLASH_SECTOR_SIZE + off),
           size);

    return 0;
}

// Program a region in a block. The block must have previously
// been erased. Negative error codes are propagated to the user.
// May return LFS_ERR_CORRUPT if the block should be considered bad.
int pico_prog(const struct lfs_config *c, lfs_block_t block, lfs_off_t off, const void *buffer, lfs_size_t size) {
    flash_range_program( flash_offs: FS_BASE_IN_FLASH + block * FLASH_SECTOR_SIZE + off,
                        data: (const uint8_t*)buffer,
                        count: size);

    return 0;
}

// Erase a block. A block must be erased before being programmed.
// The state of an erased block is undefined. Negative error codes
// are propagated to the user.
// May return LFS_ERR_CORRUPT if the block should be considered bad.
int pico_erase(const struct lfs_config *c, lfs_block_t block) {
    flash_range_erase( flash_offs: FS_BASE_IN_FLASH + block * FLASH_SECTOR_SIZE,
                      count: 1);

    return 0;
}

```

Rysunek 21: Funkcje dostępowe do pamięci Flash, wymagane w konfiguracji LittleFS

## 5 WiFi

Sterowanie systemem odbywa się za pomocą przeglądarki internetowej. W celu skorzystania z urządzenia należy skonfigurować połączenie z siecią WiFi.

## 6 Serwer HTTP

Powstała własna implementacja serwera HTTP. Obsługuje on metody GET oraz POST. Interpretuje parametry URL jak i format `application/x-www-form-urlencoded` używany w formularzach. Używa LittleFS do wysyłania statycznych plików. Nacisk został położony na wygodny interfejs do obsługi serwera.

```

HttpServer server;
server.set_cb_arg( arg: nullptr);
server.start( port: 80);
server.static_content( lfs: &lfs, fs_path: "/static");
server.on( method: Method::GET, path: "/root", callback: root);
server.on( method: Method::GET, path: "/json", callback: json_test_page);
server.on( method: Method::GET, path: "/form", callback: form_test_page);
server.on( method: Method::POST, path: "/form", callback: form_test_page);

```

Rysunek 22: Wygodny interfejs serwera HTTP

## 7 Kryptografia

Aby oferować podstawowy poziom bezpieczeństwa nasze rozwiązanie szyfruje wiadomości szyfrem RSA używając 32-bitowego klucza. Przesyłana jest także zaszyfrowana suma kontrolna do sprawdzania poprawności zdekodowanych danych. Funkcjonuje to analogicznie do podpisu cyfrowego

---

### Algorytm 1 Szyfrowanie

---

```
1: (n, d) – klucz prywatny
2: function SZYFRUJ(dlugosc, data, enc)
3:   for  $i \leftarrow 0$  to  $dlugosc - 1$  do
4:      $enc[i] \equiv data[i]^d \pmod{n}$ 
5:   end for
6: end function
```

---

---

### Algorytm 2 Deszyfrowanie

---

```
1: (n, e) – klucz publiczny
2: function ODSZYFRUJ(dlugosc, data, enc)
3:   for  $i \leftarrow 0$  to  $dlugosc - 1$  do
4:      $enc[i] \equiv data[i]^e \pmod{n}$ 
5:   end for
6: end function
```

---

Algorytm dzieli wejściowe dane na 16-bitowe bloki i szyfruje je do bloków 32-bitowych. Składowe kluczy mają po 32 bity. Wiadomość wydłuża się dwukrotnie. Teraz jej długość to 32 bajty.

## 8 Uruchomienie aplikacji

Wraz z programami dostarczony jest plik README.md w którym opisane jest jak skompilować i wgrać aplikację do systemu mikroprocesorowego.