

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Informatyka (INF)
SPECJALNOŚĆ: Inżynieria internetowa (INT)

**PRACA DYPLOMOWA
INŻYNIERSKA**

Projekt i realizacja urządzenia do zdalnej kontroli
wzmacniacza gitarowego

Design and implementation of a device for remote
control of a guitar amplifier

AUTOR:
Oleksii Pilkevych

PROWADZĄCY PRACĘ:
dr inż. Paweł Ksieniewicz PWr

OCENA PRACY:

Spis treści

1 Wstęp	3
1.1 Motywacja	3
1.2 Wymagania	3
1.3 Konkurencyjne rozwiązania	4
1.3.1 ACPAD	4
1.3.2 Livid Guitar Wing	5
1.3.3 Aalberg audio MOON MO-1	6
1.4 Wyzwania	7
1.5 Struktura pracy	7
2 Cel i zakres	9
2.1 Zakres	9
2.2 Stack technologiczny	10
2.2.1 Sprzęt	10
2.2.2 Oprogramowanie	13
3 Projekt systemu	15
3.1 Standard MIDI	15
3.1.1 Komunikaty MIDI	15
3.1.2 Złącze MIDI	16
3.2 Obsługa modułów BLE	17
3.3 REST API	17
4 Implementacja	19
4.1 Nadajnik	19
4.1.1 Schemat do zaprogramowania ATTINY85	19
4.1.2 Schemat standalone	21
4.1.3 Ustawienie modułu BLE jako slave	22
4.1.4 Obsługa przycisków	23
4.2 Odbiornik	25
4.2.1 Schemat	25
4.2.2 Zaprogramowanie	27
4.2.3 Konfiguracja WiFi	28
4.2.4 Ustawienie modułu BLE jako master	30
4.2.5 Nasłuchiwanie	32
4.2.6 Ładowanie ustawień MIDI	34
4.2.7 API	35
4.3 Konfigurator	36
4.3.1 Interpretacja ustawień MIDI	36

4.3.2 GUI	37
4.3.3 Eksportowanie	38
4.4 Podsumowanie	38
Literatura	40

Rozdział 1

Wstęp

1.1 Motywacja

Tematem pracy jest implementacja urządzenia do zdalnej kontroli wzmacniacza gitara-wego. Mimo, że są rozpowszechnione systemy bezprzewodowe do przesyłania sygnału analogowego z przetworników gitary do wzmacniacza, nadal jest potrzeba przełączania "preset-ów" (ustawień) na wzmacniaczach podczas występów na żywo. Do tego zazwyczaj są używane przełączniki nożne, które są podpięte kablem do wzmacniacza i leżą na podłodze. Minusem tego rozwiązania, jest to, że gitarzysta powinien pamiętać kiedy ma wrócić do konkretnego miejsca na scenie w celu przełączenia na przykład czystego dźwięku na przesterowany.

Proponowanym przez tą pracę rozwiązaniem jest montaż urządzenia bezprzewodowego bezpośrednio na gitarze, co umożliwia wysyłanie dowolnych komunikatów MIDI zdalnie z dowolnego miejsca na scenie (w zasięgu urządzenia bezprzewodowego). Komunikaty trafiają do odbiornika, który jest zasilany z gniazdka, albo power banku i jest podpięty kablem do wzmacniacza gitara-wego. Ponieważ protokół MIDI jest bardzo rozbudowany i uniwersalnie wspierane - daje to dużo większe możliwości niż samo przełączanie ustawień.

1.2 Wymagania

- Minimalny zasięg około 10 metrów
- Mały pobór mocy nadajnika pozwalający na wykorzystanie baterii CR2032
- Możliwość konfigurowania sekwencji komunikatów MIDI pod każdym przyciskiem
- Czas reakcji nie większy niż 500 milisekund
- Komunikacja odbiornika z wzmacniaczem przez interfejs MIDI (5-pinowe złącze DIN)

1.3 Konkurencyjne rozwiązania

1.3.1 ACPAD



ACPAD pojawił się na stronie Kickstarter w 2016 roku. Urządzenie jest skierowane wyłącznie na gitary akustyczne, ma wiele czujników piezoelektrycznych. Umożliwia zaprogramowanie różnych dźwięków pod każdy czujnik.

Mimo, że ACPAD był stworzony w celu umożliwienia grania na syntezatorach/wirtualnej perkusji podczas grania na gitarze akustycznej - ponieważ wspiera MIDI, jest możliwe również przełączanie ustawień.

Minusy:

- Wspiera wyłącznie gitary akustyczne
- Jest dużo bardziej rozbudowane niż zwykły przełącznik ustawień, co powoduje, że więcej kosztuje (około 450 euro)
- Nie ma odbiornika ze złączem MIDI tylko USB

1.3.2 Livid Guitar Wing



Urządzenie jest oparte o Bluetooth low energy. Ma wiele czujników, przystępna cena (130 dolarów). Minusy:

- Jest zaprojektowane pod jedną specyficzną formę gitary
- Nie ma odbiornika ze złączem MIDI, więc do przekierowania komunikatów MIDI do wzmacniacza byłby potrzebny laptop z zewnętrzną kartą dźwiękową, co utrudnia sprawę oraz wydłuża czas reakcji

1.3.3 Aalberg audio MOON MO-1



Głównym plusem urządzenia jest to, że ma dedykowany odbiornik. Minusy:

- Ma tylko jeden duży przycisk, reszta jest w takim miejscu, do którego jest trudno się dostać palcami, szczególnie podczas aktywnego grania
- Projekt na kickstarterze został zawieszony, więc nie ma oficjalnej możliwości zakupu

1.4 Wyzwania

Poniżej są wymienione problemy, których rozwiązanie jest niezbędne do realizacji projektu:

- Wybór urządzeń i protokołu do bezprzewodowej transmisji danych
- Wybór mikrokontrolerów do zaimplementowania całej logiki
- Zapewnienie minimalnego możliwego poboru mocy
- Zapewnienie minimalnego możliwego czasu reakcji
- Wybór stacku technologicznego z niezbędnym wsparciem
- Dostosowanie się do ograniczeń sprzętowych mikrokontrolerów
- Rozwiążanie problemu parowania urządzeń bezprzewodowych
- Dostosowanie się do wymagań interfejsu MIDI
- Stworzenie lightweight interfejsu do konfigurowania systemu
- Integracja z urządzeniami zewnętrznymi (wzmacniacz, karta muzyczna itp)

1.5 Struktura pracy

W drugiej części pracy jest opisany zakres projektu oraz wybrane rozwiązania technologiczne.

W trzeciej części opisany jest standard MIDI, komunikacja przez BLE oraz API do konfigurowania urządzenia.

W czwartej części przejdziemy do szczegółowego opisu implementacji ze schematami, kodem źródłowym, konfiguracją modułów BLE, sposobami zaprogramowania mikrokontrolerów, opisem API oraz informacjami na temat napisania GUI.

Rozdział 2

Cel i zakres

Celem projektu jest zbudowanie prototypu, który jest w stanie zademonstrować całą drogę od wciśnięcia przycisku na nadajniku do zmiany ustawienia na wzmacniaczu. Projekt nie obejmuje stworzenia płytka drukowanej oraz obudowy.

2.1 Zakres

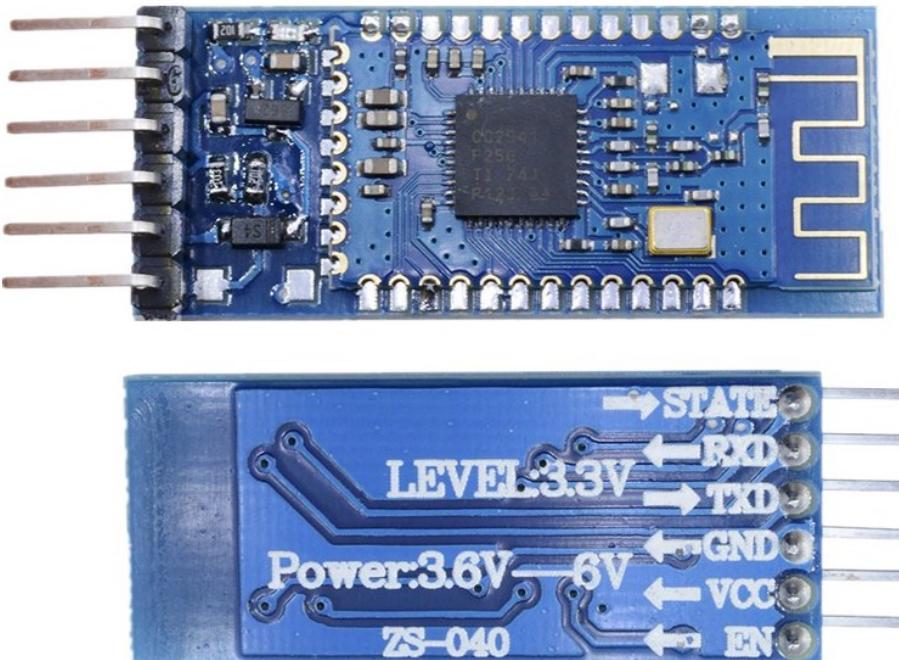
Komponenty systemu:

- Nadajnik
 - Zasilany małą baterią CR2025
 - Reaguje na wciśnięcia przycisków
 - Wysyła informację o wciśnięciu przycisku do odbiornika bezprzewodowo
- Odbiornik
 - Nasłuchuje komunikaty nadajnika
 - Ma strukturę danych która mapuje numery przycisków na konkretny komunikat MIDI (może to być sekwencja komunikatów, z której jest wybierany kolejny komunikat po każdym wciśnięciu przycisku)
 - Udostępnia interfejs do konfigurowania mapowania opisanego wyżej
 - Wysyła komunikat MIDI na interfejs DIN-5
- Aplikacja do konfigurowania odbiornika
 - Używa interfejsu udostępnionego przez odbiornik
 - Zapewnia łatwy user-friendly sposób na konfigurowanie odbiornika

2.2 Stack technologiczny

2.2.1 Sprzęt

HM-10

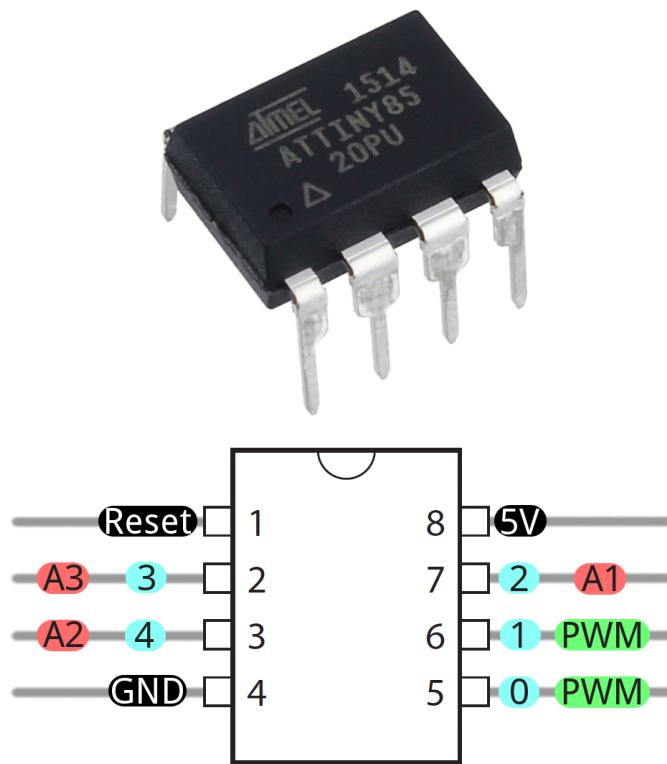


Do komunikacji bezprzewodowej został wybrany moduł BLE HM-10. Jest następcą dobrze znanego HC-05. Specyfika projektu wymaga urządzenia bezprzewodowego, które nie tylko ma mały pobór mocy, a również szybki czas reakcji. Z jednej strony mamy WiFi, które zapewnia niski czas reakcji, ale pobiera dużo prądu. Z innej strony mamy urządzenia IoT, które potrafią prawie nie zużywać prądu, ale przebywają w trybie deep sleep, wyjście z którego niestety potrzebuję czasu, na który nie możemy sobie pozwolić w przypadku tego projektu. Kompromisem jest bluetooth, a konkretnie BLE. Jest idealnym kompromisem pomiędzy poborem mocy i czasem reakcji.

Wybór był spowodowany następującymi cechami modułu:

- Najpopularniejszy jeśli chodzi o moduły BLE, więc ma dobre wsparcie
- Prosty interfejs (zasilanie, masa, piny TX/RX do transmisji)
- Obsługuje komendy AT
- Przystępna cena (około 20 zł)

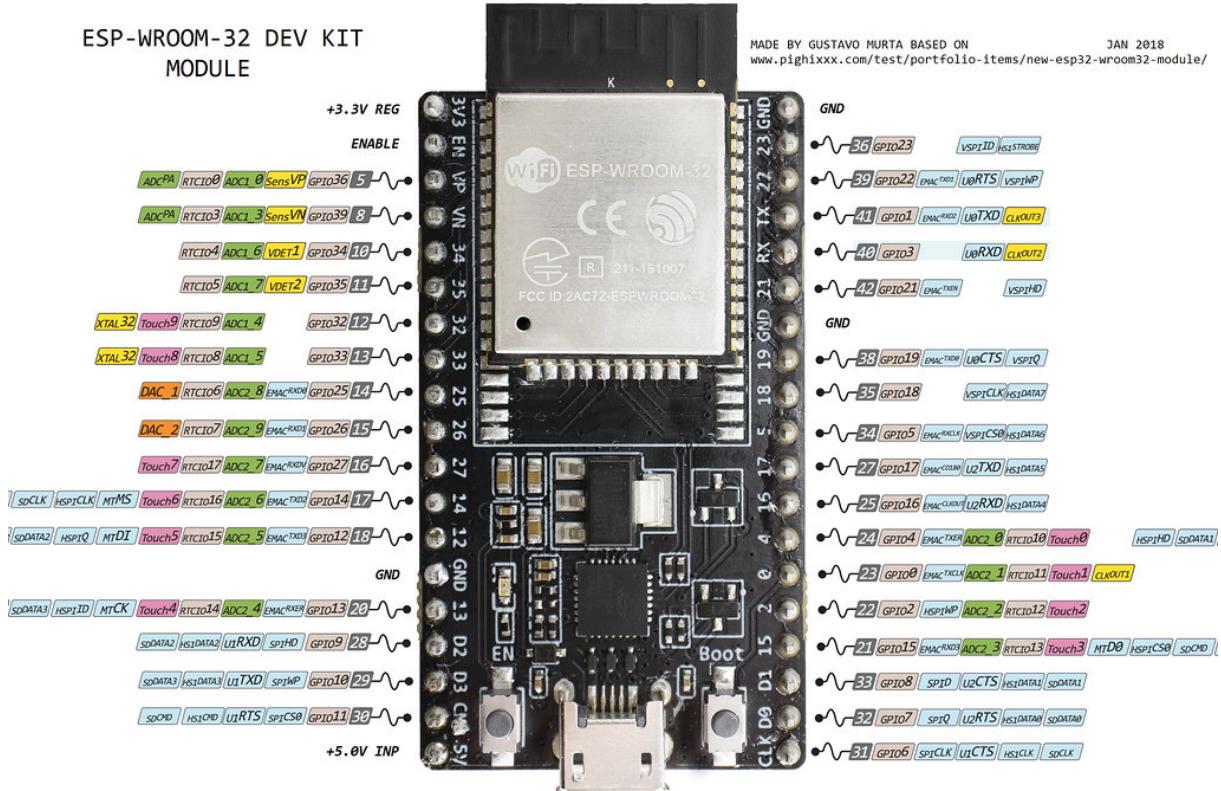
Attiny85



Do komunikacji z modułem BLE potrzebujemy mikrokontrolera. Wybór padł na ATTINY85 z następujących powodów:

- Wspiera język Arduino, więc jest łatwy w zaprogramowaniu
- Ma bardzo dobre wsparcie, więc łatwo jest znaleźć informację i się nauczyć na przykładzie innych projektów
- Pobiera bardzo mało prądu (około 4 razy mniej niż moduł BLE)
- Jest bardzo kompaktowy (rozmiarem z przycisków)
- Ma 2 piny do zasilania oraz 6 pinów I/O, czego wystarczy do komunikacji z modułem BLE oraz podpięcia czterech przycisków
- Przystępna cena (około 10 zł)

ESP-32



W przypadku odbiornika musimy zapewnić:

- Przechowywanie struktury danych z mapowaniem
- Komunikację z modułem BLE
- Komunikację z interfejsem MIDI
- Interfejs do konfigurowania

Jest dużo więcej funkcjonalności, natomiast nie ma ograniczenia na pobór mocy. Rozważana była możliwość użycia tego samego modułu BLE do interfejsu konfigurowania, natomiast to uniemożliwia jednoczesną konfigurację i testowanie przesyłu danych, ponieważ urządzenie BLE w trybie slave może trzymać tylko jedno połączenie. Prostszym rozwiązaniem jest użycie WiFi i stworzenie web serwera z REST API do konfigurowania mapowania.

Wybrany został moduł ESP-32:

- Wspiera język MicroPython, jest łatwiejszy niż mieszanka Arduino i C++, ma frameworki webowe
- Ma bardzo dobre wsparcie, jest kontynuacją ESP8266 i ma wielkie community
- Ma zintegrowane WiFi, może działać jak w trybie stacji, tak i punktu dostępu
- Przystępna cena (około 30 zł, zależy od konkretnej płyty)

2.2.2 Oprogramowanie

Nadajnik

Mikrokontroler Attiny85 jest zaprogramowany w języku Arduino z użyciem rozszerzenia w C++ do debounce-owania przycisków. Jest to niezbędne, żeby wciśnięcie przycisku nie powodowało zakłóceń w postaci wykrycia dodatkowych wciśnięć. Została użyta biblioteka SoftwareSerial do komunikacji z modułem BLE przez piny TX/RX.

Odbiornik

Odbiornik został zaprogramowany całkowicie w języku MicroPython. Do WiFi została użyta wbudowana biblioteka network. Do komunikacji z modułem BLE oraz interfejsem MIDI - biblioteka machine.UART. Do implementacji REST API do wyboru są frameworki

- Picoweb
- MicroWebSrv
- MicroWebSrv2

Podczas researchu frameworku Picoweb zauważylem problemy z zależnościami, co było znakiem słabego wsparcia. MicroWebSrv2 jest ulepszoną wersją MicroWebSrv, natomiast kiedy testowałem tą wersję frameworku - zapytania się wykonywały kilka razy dłużej.

Wybrany został MicroWebSrv dlatego, że zadowala potrzebę, i mimo, że kod nie jest najlepszej jakości (nie stosuje zasad PEP8), jest tego kodu bardzo mało (880 linijek). Co sprawia, że w przypadku braku wsparcia, naprawa problemów z frameworkiem samodzielnie jest też możliwa.

Konfigurator

Aplikacja desktopowa została napisana w języku Python w frameworku Kivy. Ten framework został wybrany, ponieważ:

- Ma bardzo dobrą dokumentację
- Out-of-the-box zapewnia responsywny layout (czego nie ma na przykład w najpopularniejszym frameworku Tkinter)
- Domyślne widgety (przyciski, napisy) już wyglądają dobrze i nie wymagają customizacji stylistycznej

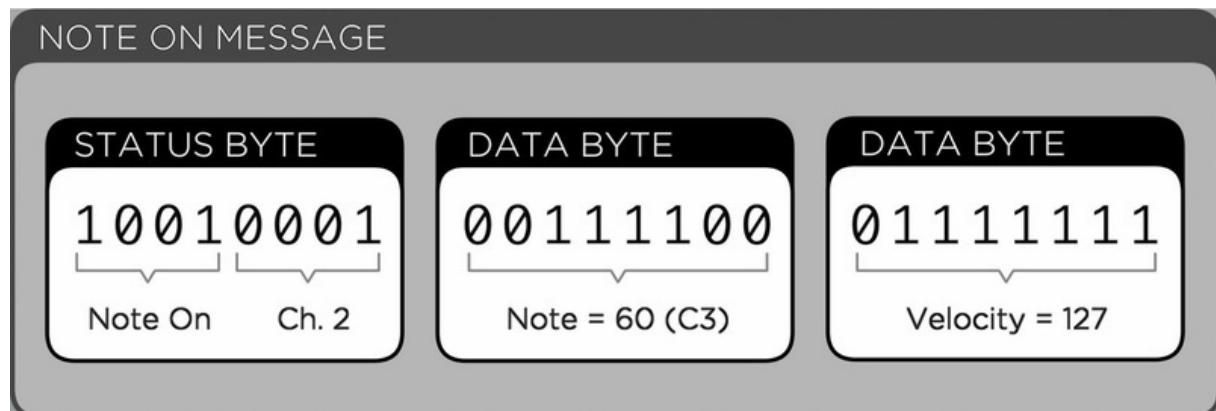
Rozdział 3

Projekt systemu

3.1 Standard MIDI

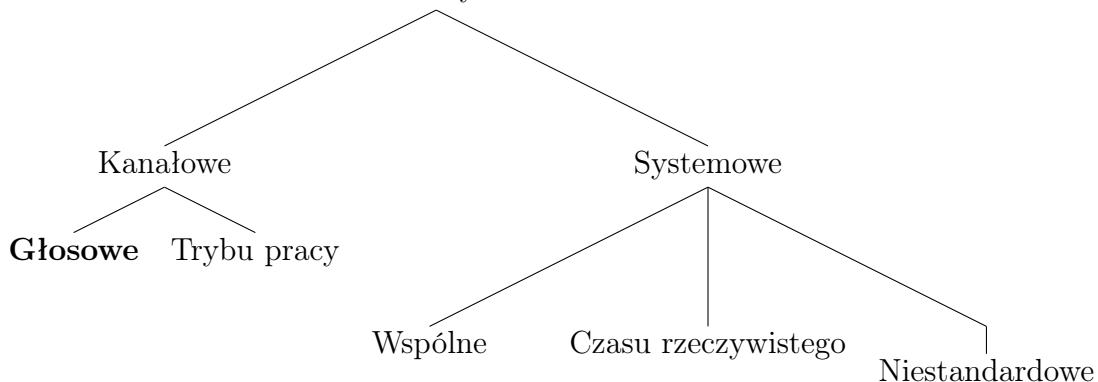
MIDI (Musical Instrument Digital Interface) - jest to standard definiujący interfejs komunikacyjny pomiędzy urządzeniami muzycznymi (komputery, karty dźwiękowe, syntezatory, wzmacniacze itp). Został stworzony w 1983 roku. Poniżej jest przedstawiona struktura przykładowego komunikatu MIDI.

3.1.1 Komunikaty MIDI



Komunikat MIDI składa się z trzech bajtów, pierwszy bajt jest bajtem statusu (określa typ wiadomości oraz kanał), kolejne dwa są bajtami danych i mają różne znaczenia w zależności od typu komunikatu. Drzewo poniżej przedstawia wszystkie możliwe typy komunikatów MIDI. W tym projekcie skupiamy się na komunikatach głosowych.

Komunikaty MIDI

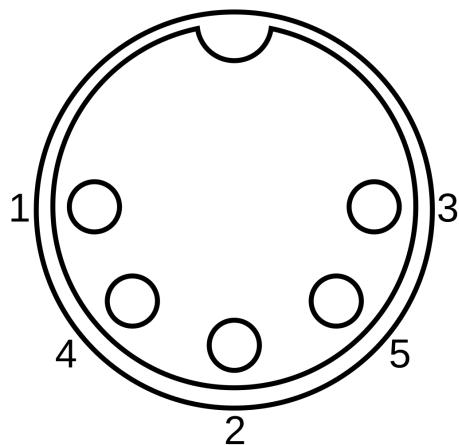


Typ komunikatu	Bajt statusu	Bajt danych 1	Bajt danych 2
Note off	8x	Key number	Note Off velocity
Note on	9x	Key number	Note on velocity
Polyphonic Key Pressure	Ax	Key number	Amount of pressure
Control Change	Bx	Controller number	Controller value
Program Change	Cx	Program number	None
Channel Pressure	Dx	Pressure value	None
Pitch Bend	Ex	MSB	LSB

Komunikaty głosowe

Do testowania najwygodniej jest wykorzystać komunikaty *Note On* oraz *Note Off* ponieważ jest łatwo monitorować te komunikaty na ścieżkach w programach muzycznych (DAW).

3.1.2 Złącze MIDI



Złącze MIDI posiada 5 pinów. Może działać jak na 3.3V tak i na 5V. Dane są przesyłane szeregowo z baud rate-m 31,250.

1. Nieużywany
2. GND
3. Nieużywany
4. Dane
5. VCC

3.2 Obsługa modułów BLE

Moduły BLE są konfigurowane za pomocą komend AT (albo inaczej "*Hayes command set*"). Zbiór tych komend jest trochę specyficzny, ponieważ projekcie został użyty klon klonu oryginalnego modułu HM-10 **MLT-BT05**. Całą listę komend można znaleźć tutaj.

Interesują nas natomiast:

Opis	Komenda	Parametr
Ustawienie trybu	AT+ROLE<Param>	0 - slave, 1 - master
Sprawdzenie wersji	AT+VERSION	N/A
Ustawienie nazwy	AT+NAME<Param>	<nazwa>
Restart modułu	AT+RESET	N/A
Wyszukiwanie modułów	AT+INQ<Param>	1 - zacząć skan, 0 - skończyć skan
Nawiązanie połączenia	AT+CONN<Param>	0-7
Tryb startowy	AT+IMME<Param>	0 - start w trybie komend, 1 - start w trybie normalnym
Włączenie trybu transmisji	AT+START	N/A
Ustawienia notyfikacji	AT+NOTI<Param>	1 - wysyłaj notyfikacje, 0 - nie wysyłaj
Reset do ustawień fabrycznych	AT+DEFAULT	N/A

3.3 REST API

Struktura danych do przechowywania mapowań przycisków do sekwencji komunikatów wygląda następująco:

```
{
  "notes": [
    [
      [
        [
          200,
          100,
          100
        ],
        [
          128,
          123,
          150
        ]
      ],
      [[234, 234, 234]],
      []
    ]
  }
}
```

Na serwerze jest wystawiony jeden endpoint wspierający zapytania GET do pobrania konfiguracji oraz POST do nadpisywania.

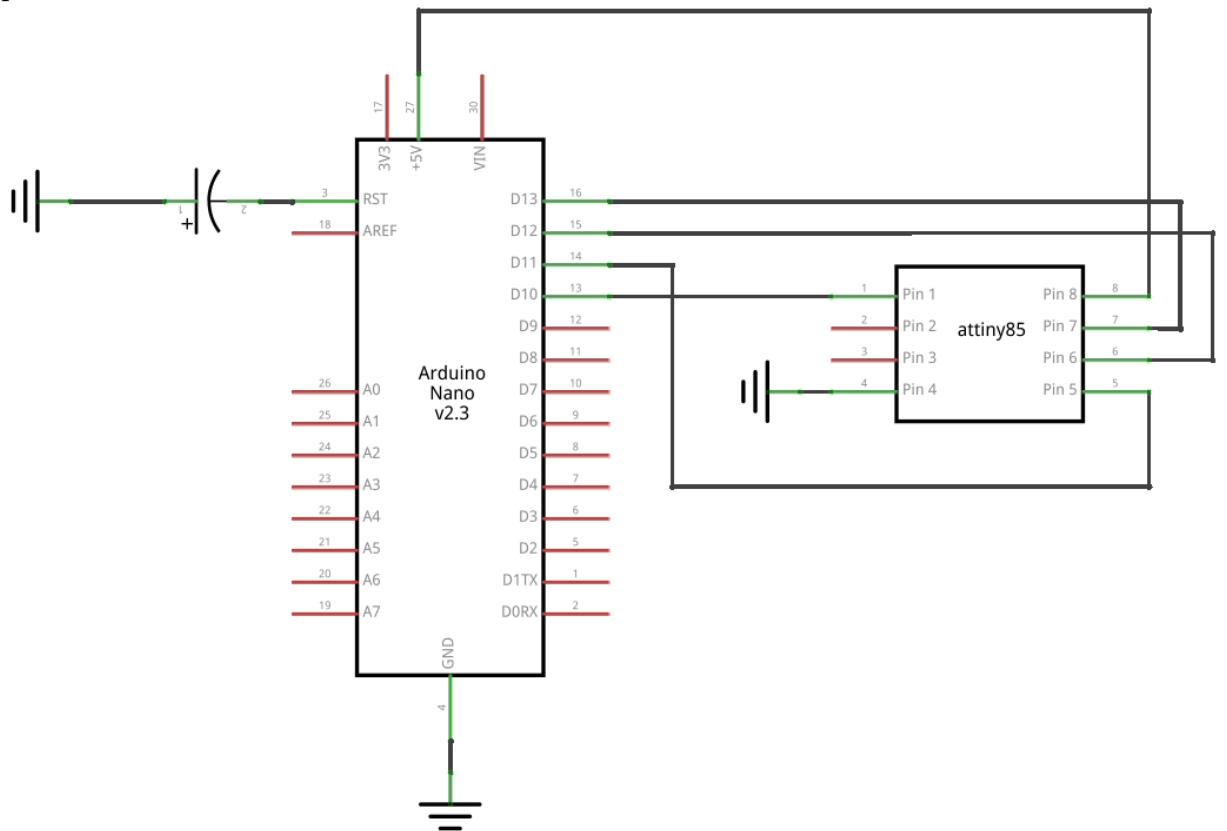
Rozdział 4

Implementacja

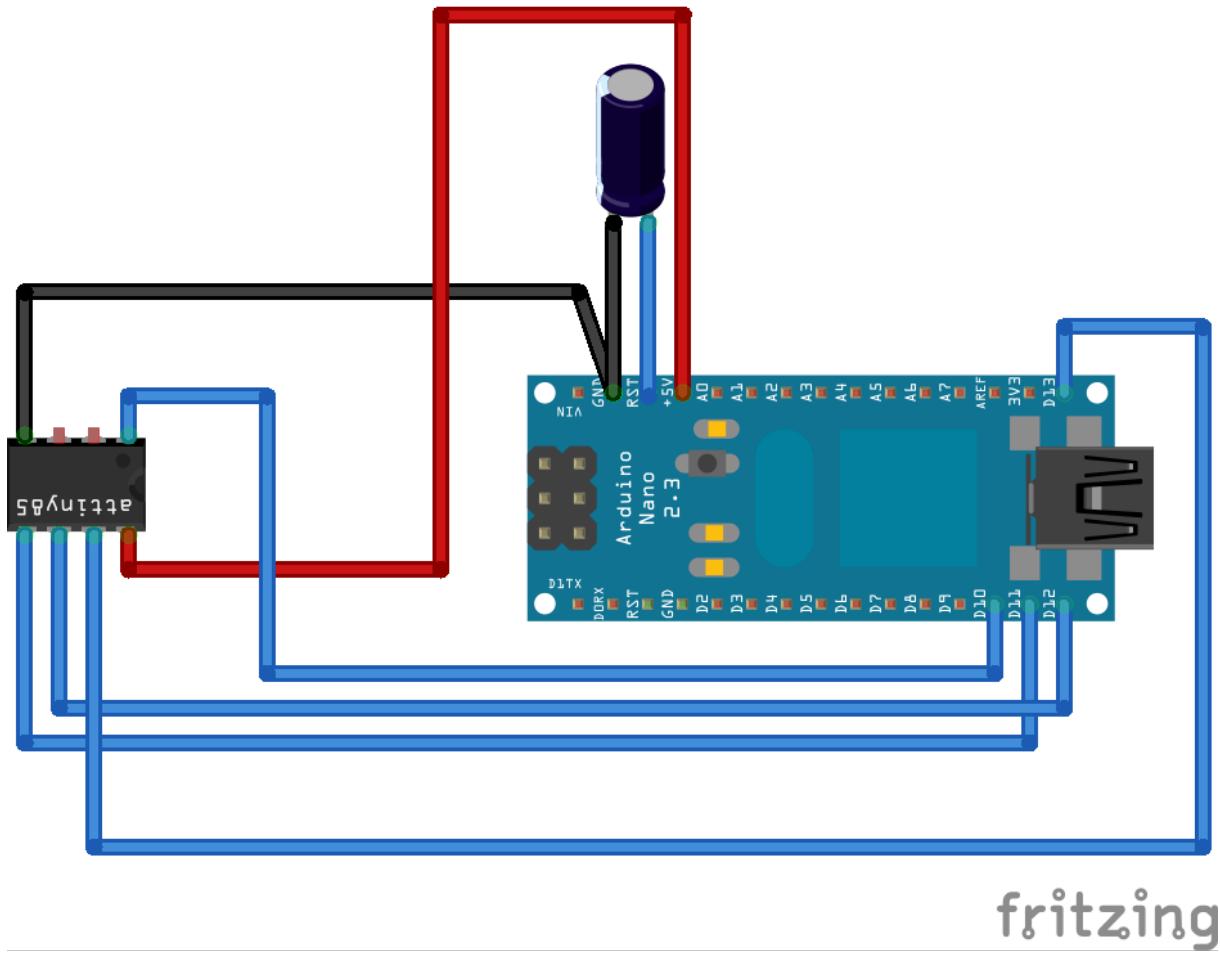
4.1 Nadajnik

4.1.1 Schemat do zaprogramowania ATTINY85

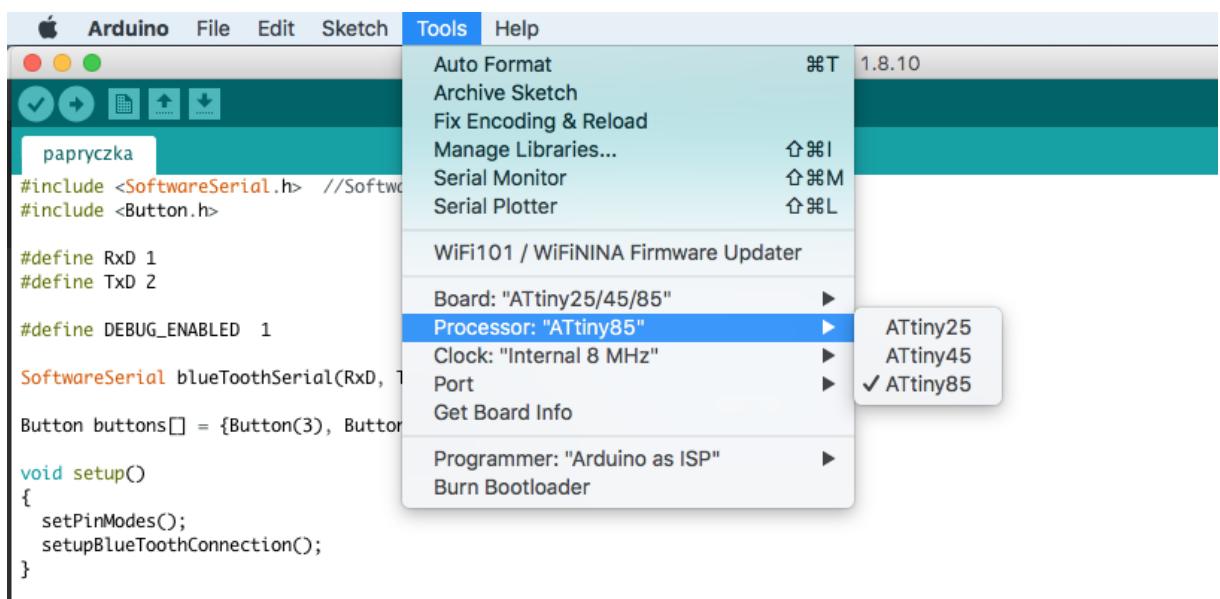
Przedstawiony poniżej schemat pokazuje jak za pomocą Arduino Nano zaprogramować chip ATTINY85.



fritzing

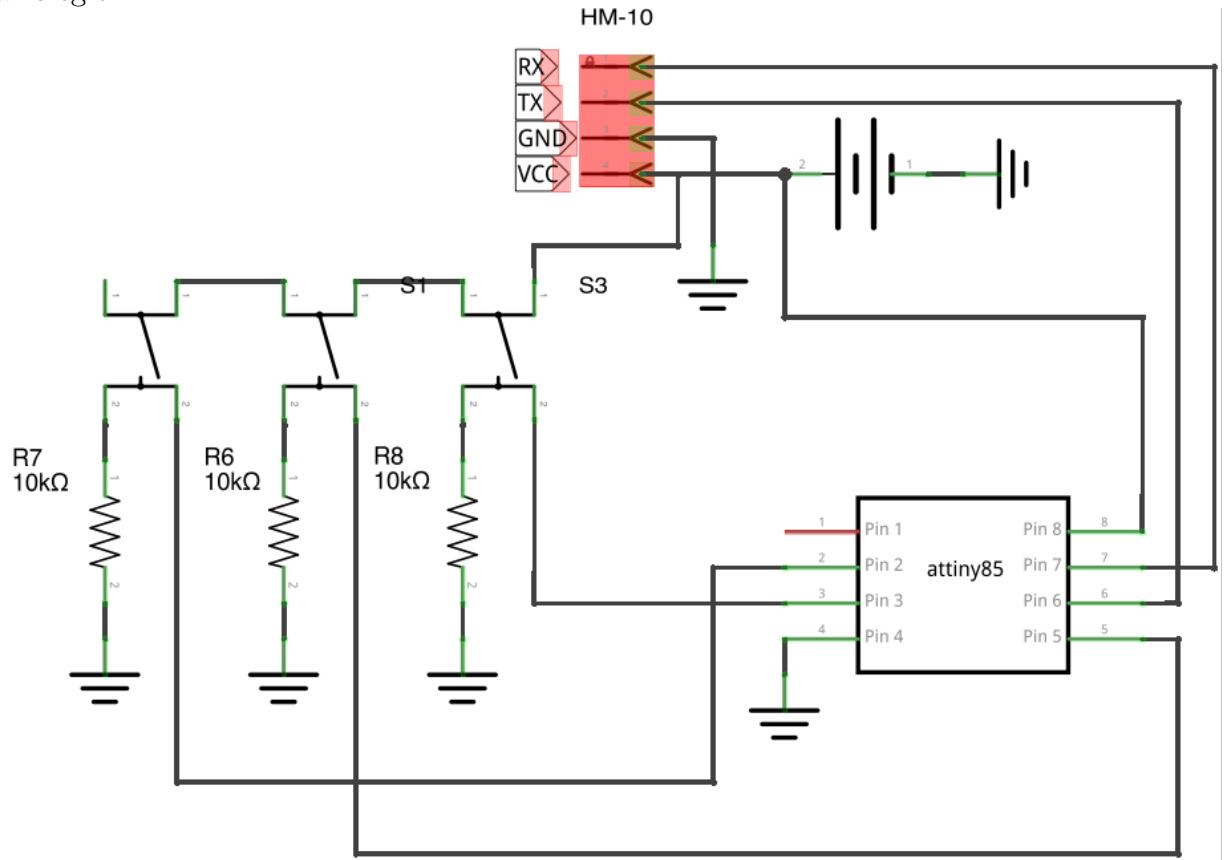


Do zaprogramowania ATTINY również niezbędne są odpowiednie ustawienia w Arduino IDE. Należy pobrać definicje mikrokontrolerów serii ATTINY, wybrać odpowiedni model (w naszym przypadku ATTINY85), ustawić zegar na 8 MHz.

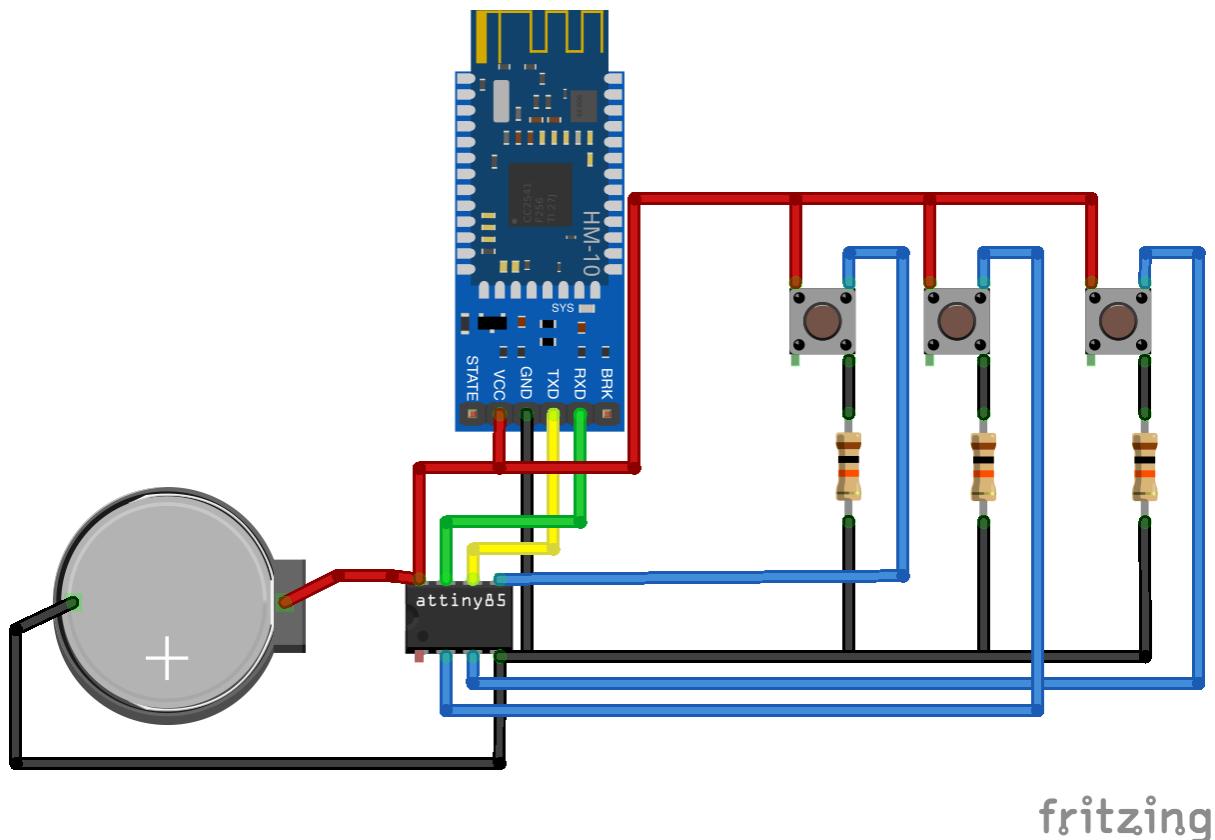


4.1.2 Schemat standalone

Przedstawiony poniżej schemat pokazuje jak połączyć zaprogramowany chip ATTINY z modułem BLE i trzema przyciskami. Warto również zauważać, że system jest bardziej stabilny (rzadziej traci połączenie) jeśli użyjemy dwóch baterii CR2032 podłączonych równolegle.



fritzing



4.1.3 Ustawienie modułu BLE jako slave

Przedstawiona poniżej funkcja jest użyta do konfigurowania modułu BLE na nadajniku jako slave. Moduł jest resetowany do ustawień fabrycznych, dostaje przypisaną nazwę, ustawia się w tryb slave z startowaniem w trybie komend z włączonymi notyfikacjami.

```

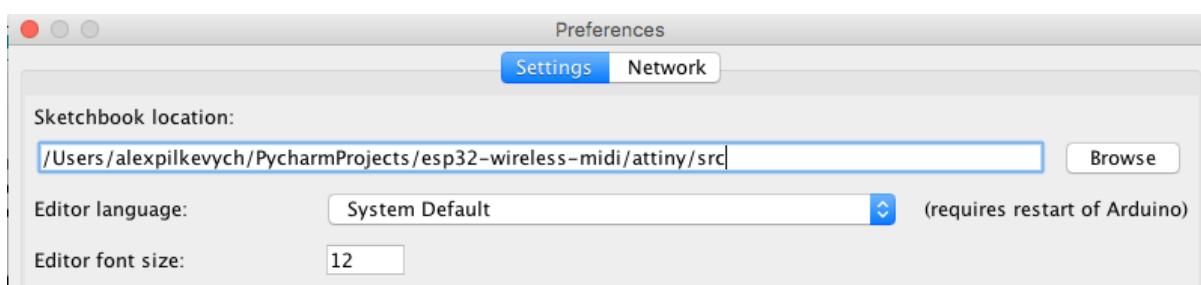
1 void setupBlueToothConnection()
2 {
3     blueToothSerial.begin(9600);
4     write("DEFAULT");
5     write("RESET");
6     write("NAMEpapryczka");
7     write("ROLE0");
8     write("IMME0");
9     write("RESET");
10    write("NOTI1");
11    blueToothSerial.flush();
12 }
```

4.1.4 Obsługa przycisków

Z wysokiego poziomu obsługa przycisków wygląda następująco. Definiujemy trzy przyciski wskazując piny I/O na ATTINY85. Później w nieskończonej pętli programu cały czas sprawdzamy stan przycisków. Jeśli jest wcisnięty - wysyłamy przez bluetooth numer przycisku.

```
1 Button buttons[] = {Button(3), Button(0), Button(4)};  
2  
3 ...  
4  
5 void loop() {  
6     for (int i = 0; i < 3; i++) {  
7         checkButton(&buttons[i], i);  
8     }  
9 }  
10  
11 void checkButton(Button* button, int message) {  
12     if(button->check(blueToothSerial)) {  
13         blueToothSerial.print(message);  
14     }  
15 }
```

Typowym problemem z implementacją przycisków jest tak zwany "debounce". Chodzi o to, że podczas wcisnięcia przycisku jest generowany szum w postaci szybko zmieniających się wartości logicznych. Logika, która rozwiązuje ten problem została wyniesiona do biblioteki *Button.h*. Przed komplikacją programu należy upewnić się, że w ustawieniach Arduino IDE jest podana właściwa ścieżka do bibliotek zewnętrznych.



Problem został rozwiązany w następujący sposób. Jeśli przycisk ma stan logiczny 1 - sprawdzamy czy od ostatniego wcisnięcia minęło 10 milisekund. Jeśli tak - raportujemy wcisnięcie, jeśli nie - traktujemy to jako szum i ustawiamy aktualny czas jako czas ostatniego wcisnięcia.

```
Button :: Button( int pin )
{
    pinMode( pin , INPUT );
    this->_pin = pin ;
}

bool Button :: check( SoftwareSerial bt )
{
    int reading = digitalRead( this->_pin );

    if ( reading == this->lastButtonState ) {
        return false ;
    }

    bool isNoise = (( millis() - this->lastPress ) < this->debounceDelay );

    this->lastPress = millis();
    this->lastButtonState = reading;

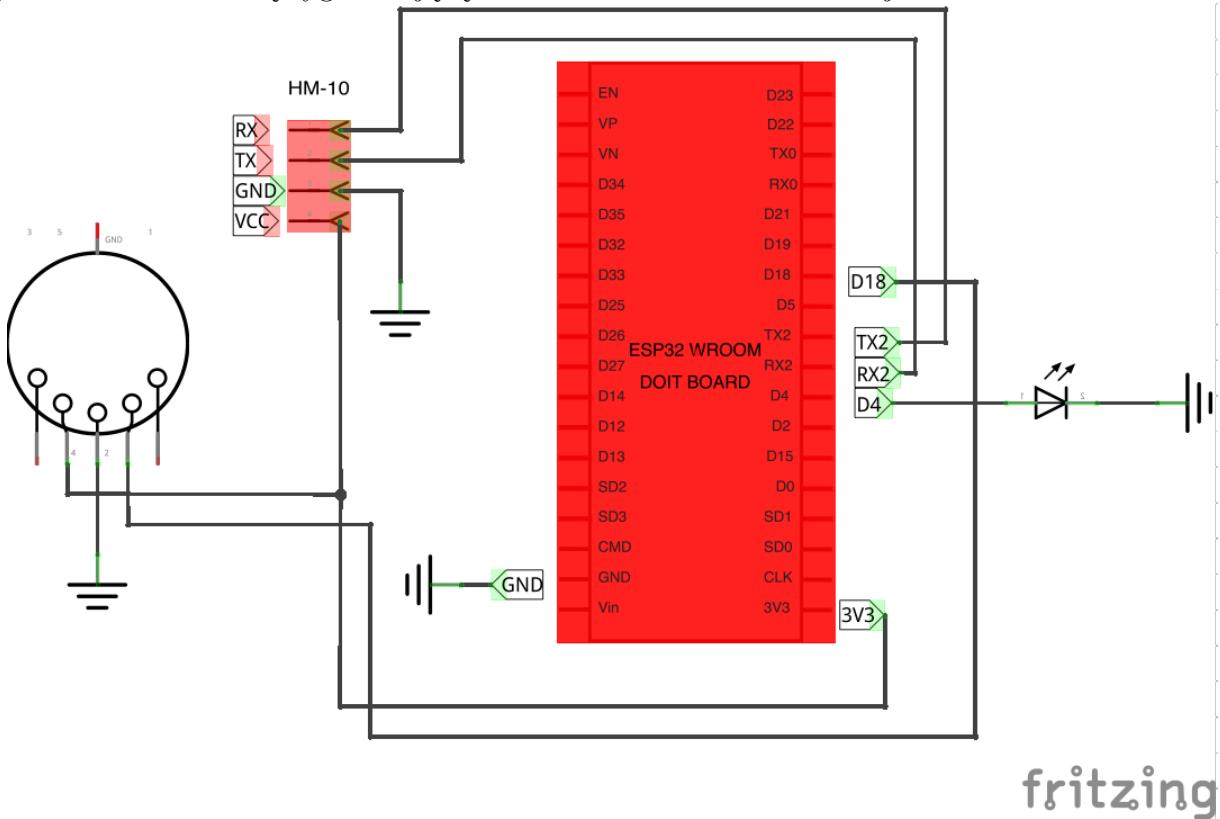
    bool buttonPressed = !isNoise && reading == HIGH;

    return buttonPressed ;
}
```

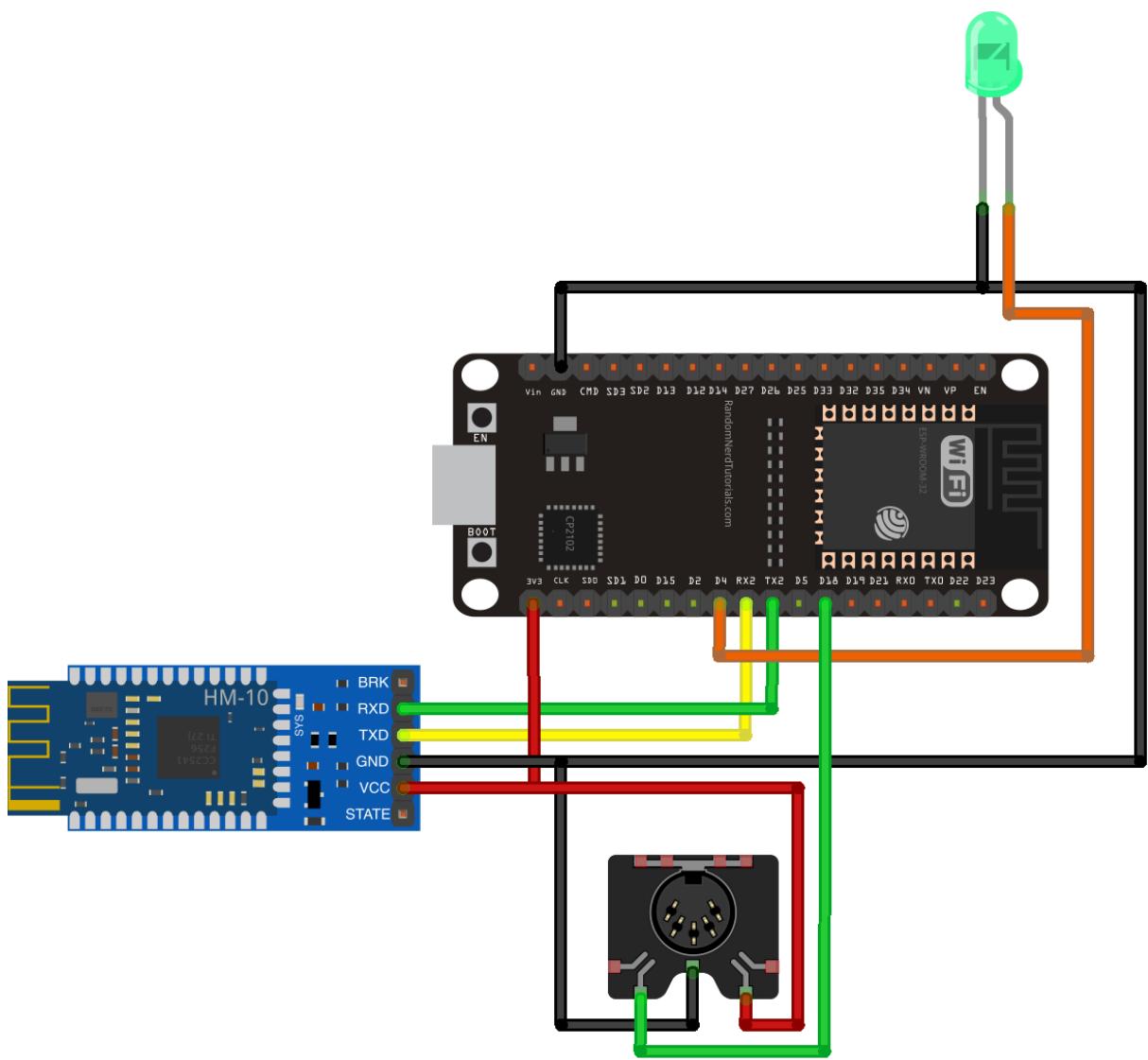
4.2 Odbiornik

4.2.1 Schemat

Przedstawiony poniżej schemat pokazuje jak podłączyć do modułu ESP-32 moduł BLE, złącze MIDI oraz diodę sygnalizującą odebranie komunikatu z nadajnika.



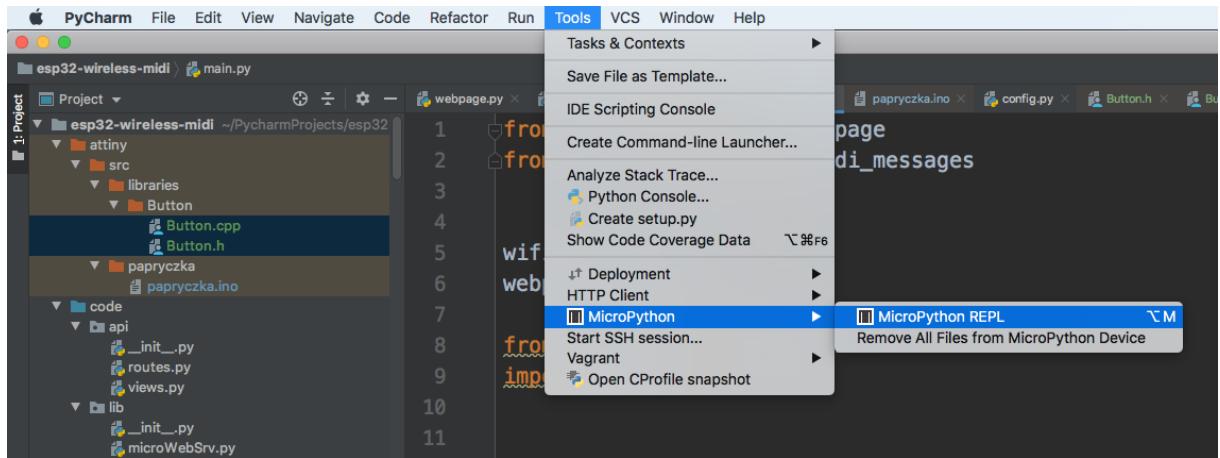
fritzing



fritzing

4.2.2 Zaprogramowanie

Żeby zaprogramować moduł ESP-32 w języku Python - należy wgrać na płytę interpreter. Po czym można się podłączyć do konsoli Pythonowej przez (na przykład) wtyczkę MicroPython dla PyCharm IDE.



Przykładowe logi aplikacji odbiornika wyglądają tak:

```
MicroPython v1.11-396-g27fe84e66 on 2019-10-05; ESP32 module with ESP32
Type "help()" for more information.
>>>
MPY: soft reboot
I (23183) event: sta ip: 192.168.1.27, mask: 255.255.255.0, gw: 192.168.1.1
I (23183) network: GOT_IP
b'+DEFAULT\r\nOK\r\n'
b'+RESET\r\nOK\r\n'
b'+IMME=1\r\nOK\r\n'
b'+ROLE=1\r\nOK\r\n'
b'+RESET\r\nOK\r\n'
b'OK\r\n'
b'OK\r\nn+INQS\r\nn+INQ:1 0x001583F0239E -74\r\n'
b'+INQCV\r\nn+Connecting 0x001583F0239E\r\nn+INQE\r\nnDevices Found 1\r\nr\r\nn+Connected 0x001583F0239E\r\n
I (31223) uart: ALREADY NULL
b'0'
Sent [128, 123, 150]
Pressed 0
b'0'
Sent [200, 100, 100]
Pressed 0
b'1'
Sent [234, 234, 234]
Pressed 1
```

Należy uważać, ponieważ płytka potrafi obsługiwać tylko jedno połączenie. Jeśli nie wyłączymy terminal to nie będzie można wgrać plik.

Podczas startu interpretera Pythona automatycznie odpala plik *main.py*. Ten plik jest punktem wejściowym do aplikacji.

Zarządzać plikami można (na przykład) przez narzędzie konsolowe *ampy*. Pozwala ono na wgrywanie, pobieranie oraz usuwanie plików/katalogów z płytka.

```
[alex-s-imac:Desktop alexpilkevych$ ampy --port /dev/cu.SLAB_USBtoUART ls
./DS_Store
/code
/load_testing
/main.py
[alex-s-imac:Desktop alexpilkevych$ ampy --port /dev/cu.SLAB_USBtoUART rm main.py
[alex-s-imac:Desktop alexpilkevych$ ampy --port /dev/cu.SLAB_USBtoUART ls
./DS_Store
/code
/load_testing
alex-s-imac:Desktop alexpilkevych$ ]
```

4.2.3 Konfiguracja WiFi

ESP-32 wspiera jak tryb Access Point (punkt dostępu) tak i Station (tryb stacji). W środowisku testowym podłączenie płytka jako stacji do sieci domowej pozwala na korzystanie z internetu w trakcie testowania aplikacji. W środowisku docelowym musimy jednak zapewnić tryb punktu dostępu, ponieważ urządzenie powinno być przenośne i niezależne od innych sieci. Z tych powodów obydwa tryby zostały zaimplementowane. Da się je przełączyć odpowiednio ustawiając zmienną *ACCESS_POINT_MODE*. W obu przypadkach używamy tego samego adresu IP, ponieważ jest zahardkodowany w aplikacji konfigurującej odbiornik.

```
ACCESS_POINT_MODE = True
```

```
class AccessPointConfig :
    SSID = 'ESP-32-WOOOWOOWOO'
    PASSWORD = 'thisisnotaspoon'
    IP = '192.168.1.27'
    SUBNET_MASK = '255.255.255.0'
    GATEWAY = '192.168.1.27'

class StationConfig :
    SSID = '<SSID>'
    PASSWORD = '<PASSWORD>'
    IP = '192.168.1.27'
    SUBNET_MASK = '255.255.255.0'
    GATEWAY = '192.168.1.1'
```

```
import network
from code import config

def connect_station(c: config.StationConfig):
    station = network.WLAN(network.STA_IF)
    station.active(True)
    station.connect(c.SSID, c.PASSWORD)
    station.ifconfig((c.IP, c.SUBNET_MASK, c.GATEWAY, '8.8.8.8'))

def connect_access_point(c: config.AccessPointConfig):
    ap = network.WLAN(network.AP_IF)
    ap.active(True)
    ap.config(essid=c.SSID, password=c.PASSWORD)
    ap.ifconfig((c.IP, c.SUBNET_MASK, c.GATEWAY, '8.8.8.8'))

def start_wifi():
    if config.ACCESS_POINT_MODE:
        connect_access_point(config.AccessPointConfig())
    else:
        connect_station(config.StationConfig())
```

4.2.4 Ustawienie modułu BLE jako master

Poniżej jest przedstawiona klasa, odpowiedzialna za ustawienie modułu BLE w trybie master i nasłuchiwanie.

```
class Receiver:

    def connect(self):
        self.factory_default()
        self.reset()
        self.set_imme()
        self.set_master()
        self.reset()
        self.start()
        self.inquire()
        self.connect_first()
        ...

    def run_command(self, command):
        self.uart.write(command + '\r\n')
        time.sleep(1)
        return self.wait()

    def reset(self):
        return self.run_command('AT+RESET')

    def factory_default(self):
        return self.run_command('AT+DEFAULT')

    def set_master(self):
        return self.run_command('AT+ROLE1')

    def set_imme(self):
        return self.run_command('AT+IMME1')

    def start(self):
        return self.run_command('AT+START')

    def inquire(self):
        return self.run_command('AT+INQ')

    def connect_first(self):
        return self.run_command('AT+CONN1')

    def wait(self):
        data = None
        while not data:
            data = self.read()
        print(data)
        return data
```

Żeby tą klasę użyć wystarczy odpalić:
Receiver().connect()

Wtedy są wykonywane następujące działania:

1. Utworzenie połączenia szeregowego z modułem BLE
2. Reset do ustawień fabrycznych
3. Restart modułu
4. Ustawienie trybu komend
5. Ustawienie trybu master
6. Restart modułu
7. Włączenie trybu transmisji
8. Skanowanie urządzeń
9. Nawiązanie połączenia

W tym momencie mamy sparowane moduły BLE, można zauważyć, że czerwona dioda na modułach przestaje mrugać i stabilnie się świeci.

4.2.5 Nasłuchiwanie

W momencie kiedy połączenie pomiędzy modułami BLE na nadajniku i odbiorniku zostało nawiązane, wykonywane są następujące czynności:

1. Utworzenie połączenia szeregowego z interfejsem MIDI
2. Początek nasłuchiwania w pętli nieskończonej.
 - W przypadku komunikatu *Disconnected* wywołujemy próbę nawiązania nowego połączenia w nowej pętli nieskończonej
 - W przypadku liczby od 0-2 jest triggerowane wysyłanie następnego w sekwencji komunikatu MIDI dla danego przycisku

```
class MidiInterface:  
    baud = 31250  
  
    def __init__(self):  
        self.uart = None  
  
    def connect(self):  
        self.uart = UART(2, self.baud)  
        self.uart.init(self.baud, bits=8, parity=None, stop=1, tx=18)  
  
    def send(self, slot):  
        if midi_messages.slots.messages[slot]:  
            message = midi_messages.slots.pop_message(slot)  
            self.uart.write(bytes(message))  
            print('Sent {}'.format(message))
```

```
class Receiver:
    def __init__(self):
        ...

        self.EVENTS = {
            b'Disconnected': self.reconnect,
            b'0': lambda: self.on_button(0),
            b'1': lambda: self.on_button(1),
            b'2': lambda: self.on_button(2)
        }
        self.midi = MidiInterface()

    def connect(self):
        ...
        self.midi.connect()
        self.loop()

    def on_button(self, slot):
        self.led.on()
        self.midi.send(slot)
        self.led.off()
        print('Pressed_{}'.format(slot))

    def reconnect(self):
        while True:
            data = self.connect_first()
            if b'Connected' in data:
                return self.loop()
            time.sleep(1)

    def read(self):
        return self.uart.read()

    def wait(self):
        data = None
        while not data:
            data = self.read()
        print(data)
        return data

    def loop(self):
        while True:
            data = self.wait()
            for expected, action in self.EVENTS.items():
                if expected in data:
                    action()
```

4.2.6 Ładowanie ustawień MIDI

Mapowanie przycisków do sekwencji komunikatów jest przechowywane w pliku JSON. Podczas startu aplikacji jest tworzony singleton klasy *Slots*, pozwalający na iterowanie po komunikatach przy wcisnięciu przycisków na nadajniku (metoda *pop_message*), ładowanie i ustawianie mapowań w runtime przez API (metody *get* i *set*).

```
class Slots:

    quantity = 3

    def __init__(self):
        self.indexes = {slot: 0 for slot in range(self.quantity)}
        self.messages = [[] for slot in range(self.quantity)]

    @validate_slot
    def pop_message(self, slot):
        index = self.indexes[slot]
        messages = self.messages[slot]
        index = (index + 1) % len(messages)
        self.indexes[slot] = index
        return messages[index]

    @validate_slot
    def get(self, slot):
        return self.messages[slot]

    @validate_slot
    def set(self, slot, messages):
        self.indexes[slot] = 0
        self.messages[slot] = messages


class MidiMessages:

    config_json_path = '/code/config.json'

    def __init__(self):
        self.slots = Slots()

    def load(self):
        with open(self.config_json_path) as config_json:
            data = ujson.load(config_json)
            for slot, messages in enumerate(data['notes']):
                self.slots.set(slot, messages)

    def save(self):
        with open(self.config_json_path, 'w+') as config_json:
            ujson.dump({'notes': self.slots.messages}, config_json)
```

4.2.7 API

API zostało napisane na bazie frameworku MicroWebSrv. Jest maksymalnie proste i kompaktowe. Dane są w formacie "lista list list", gdzie główną listą jest lista przycisków, każdy przycisk to lista komunikatów MIDI, każdy komunikat to lista z trzech liczb (ponieważ komunikat MIDI składa się z trzech bajtów). Tak wygląda przykładowa zwrotka z zapytania GET:



```
[[[145, 100, 100], [129, 100, 100]], [[145, 102, 100], [129, 102, 100]], [[145, 103, 100], [129, 103, 100]]]
```

Cały kod API to mapowanie URL do widoków oraz widoki, używające interface singletona, opisanego w poprzedniej sekcji.

```
from . import views

route_handlers = [
    ('/notes/', 'GET', views.get_notes),
    ('/notes/', 'POST', views.set_notes)
]

...

from code.config import midi_messages

def get_notes(http_client, http_response):
    messages = midi_messages.slots.messages
    http_response.WriteResponseJSONOK(messages)

def set_notes(http_client, http_response):
    slots = http_client.ReadRequestContentAsJSON()
    for slot, messages in enumerate(slots):
        midi_messages.slots.set(slot, messages)
    midi_messages.save()
    http_response.WriteResponseOk()
```

4.3 Konfigurator

4.3.1 Interpretacja ustawień MIDI

Można zauważyć, że ani nadajnik ani odbiornik w żaden sposób nie interpretują komunikatów MIDI. Te komponenty systemu nie mają pojęcia co one przesyłają, po prostu widzą, że mają sprawę z trzema bajtami. Natomiast po stronie konfiguratora te dane musimy przetłumaczyć na język ludzi. Poniżej jest przedstawiona część kodu GUI, wykonująca to zadanie.

```
MESSAGE_TYPES = {
    0b1001: 'note_on',
    0b1000: 'note_off',
    0b1010: 'polyphonic',
    0b1011: 'cc',
    0b1100: 'pc',
    0b1101: 'channel_pressure',
    0b1110: 'pitch_bend'
}
MESSAGE_TYPE_TO_BYTE = {v: k for k, v in MESSAGE_TYPES.items()}

class MidiMessage:

    def __init__(self, status, data_1=None, data_2=None):
        self.status = status
        self.data_1 = data_1
        self.data_2 = data_2

    @classmethod
    def from_form(cls, _type, channel=None, data_1=None, data_2=None):
        status = MESSAGE_TYPE_TO_BYTE[_type] * 2 ** 4 + int(
            channel)
        return cls(status, int(data_1), int(data_2))

    @property
    def type(self):
        type_byte = self.status // 2 ** 4
        return MESSAGE_TYPES[type_byte]

    @property
    def channel(self):
        channel_byte = self.status - (self.status // 2 ** 4) * 2
        ** 4
        return channel_byte

    @property
    def raw(self):
        return [self.status, self.data_1, self.data_2]
```

4.3.2 GUI

GUI zostało napisane w frameworku Kivy. Poniżej są przedstawione przykładowe widgety do reprezentowania ustawień MIDI.

```
Spinner(text=message_type or '<Select>', values=MESSAGE_TYPES.  
values())  
Spinner(text=str(channel) or '1', values=(str(i) for i in range  
(1, 17)))  
TextInput(multiline=False, input_type='number', text=str(data_2)  
)  
Button(text='Delete')
```

Komunikacja z API jest realizowana przez bibliotekę *requests*.

```
def get_slots():  
    response = requests.get('http://192.168.1.27/notes/').text  
    return json.loads(response)  
  
def set_slots(slots):  
    slots = [[m.raw for m in messages] for messages in slots]  
    return requests.post('http://192.168.1.27/notes/', json=slots)
```

Przykładowa konfiguracja none_on/note_off per przycisk wygląda następująco:

Midi				
Slot 0	Type	Channel	Data 1	Data 2
Delete	note_on	1	100	100
Delete	note_off	1	100	100
Add message				
Slot 1	Type	Channel	Data 1	Data 2
Delete	note_on	1	102	100
Delete	note_off	1	102	100
Add message				
Slot 2	Type	Channel	Data 1	Data 2
Delete	note_on	1	103	100
Delete	note_off	1	103	100
Add message				
Update				

4.3.3 Eksportowanie

Kivy pozwala na eksportowanie aplikacji desktopowej oraz mobilnej. Zostało przetestowane eksportowanie dla systemu Mac OS. Wymaga zainstalowania biblioteki *pyinstaller* oraz odpalenia komendy:

```
pyinstaller --onefile --clean --windowed --name touchtracer --exclude-module _tkinter --exclude-module Tkinter --exclude-module enchant --exclude-module twisted main.py
```

4.4 Podsumowanie

Projekt udało się zrealizować. Moduł HM-10 potrafi z zadowalającą szybkością przesyłać informację o wciśniętych przyciskach. Moduł ESP-32, zaprogramowany w języku MicroPython radzi sobie z jednaczesnym serwaniem API, oraz komunikacją z modułem BLE i złączem MIDI poprzez UART.

Znane problemy:

- Bardzo trudno było znaleźć dokumentację do wykorzystanego klonu modułu HM-10. Większość komend trzeba było znajdować metodą prób i błędów. Problem można rozwiązać jedynie kupując droższy oryginalny moduł.
- Proces parowania modułów BLE w tym projekcie jest uproszczony i ma wady. Nie ma żadnego sprawdzenia, czy zeskanowany moduł jest nadajnikiem, połączenie jest realizowane automatycznie z najbliższym modułem BLE w zasięgu odbiornika. Żeby ten problem rozwiązać można by było dodać przycisk na odbiorniku oraz skonfigurować jeden z przycisków na nadajniku tak, żeby parowanie się odbywało tylko przy wcięciu obu przycisków jednocześnie. Alternatywnie można sprawdzać adres MAC zeskanowanego urządzenia.
- Nie ma możliwości zmiany ustawień punktu dostępu z poziomu użytkownika.
- Czasami nadajnik traci połączenie z odbiornikiem. Żeby tego uniknąć zaleca się podpinać kilka baterii CR2032 równolegle.
- Czasami nie działa automatyczny reconnect w przypadku wyłączenia nadajnika. W takim przypadku jedynym rozwiązaniem jest restartowanie odbiornika.

Kod projektu:

<https://github.com/alexpilk/esp32-wireless-midi>.

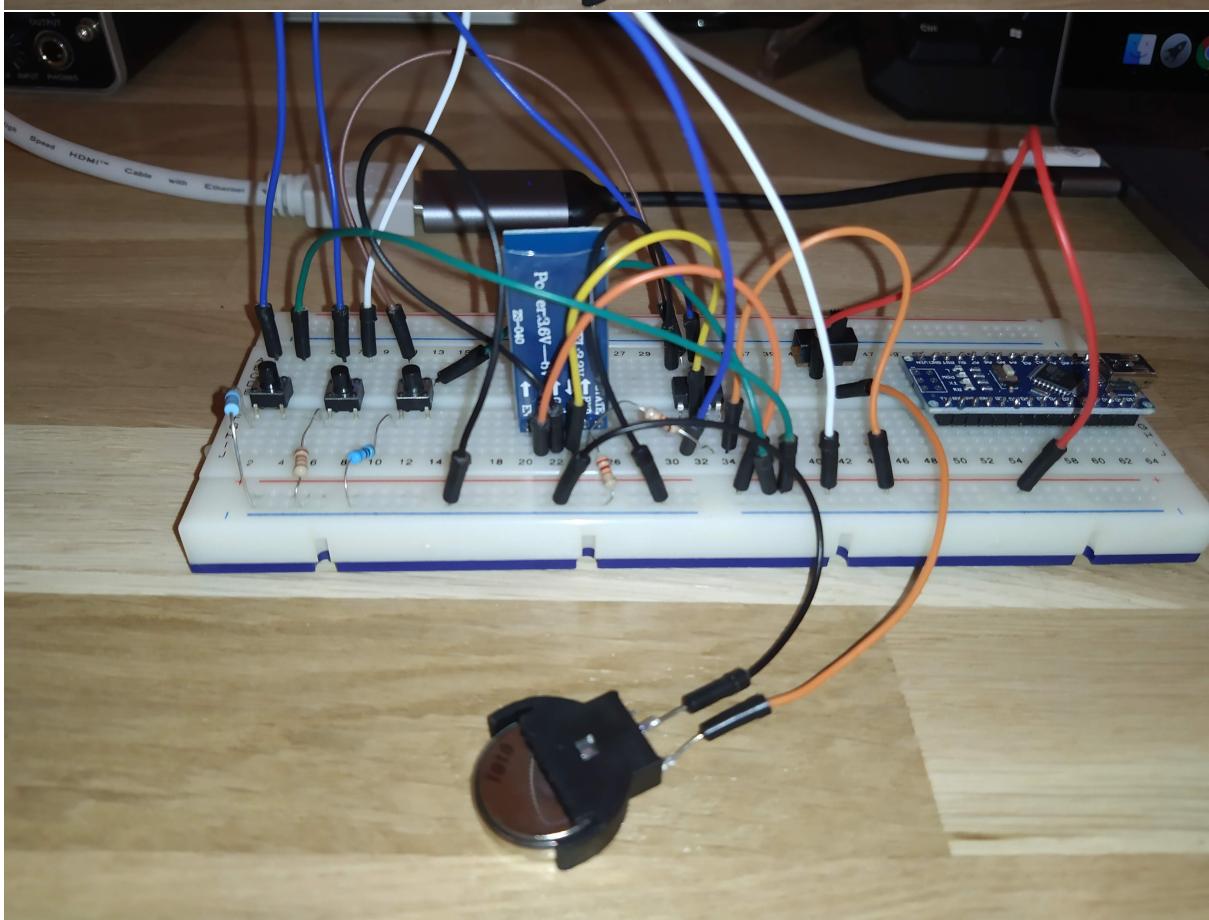
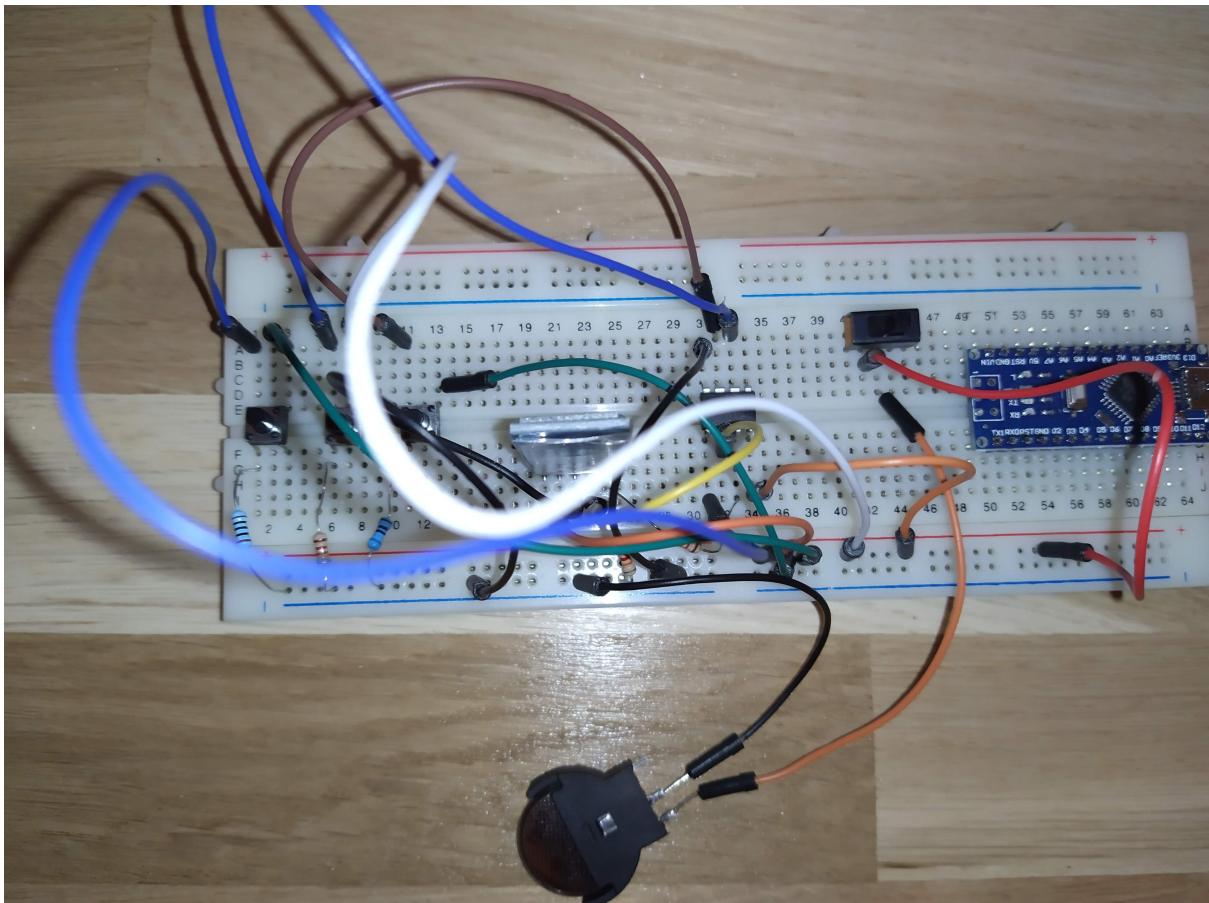
Na tym filmie odbiornik jest podłączony przez złącze MIDI do karty dźwiękowej. Sygnał z wejścia jest odbierany przez program Reaper, a nuty MIDI są przepuszczane przez syntezator:

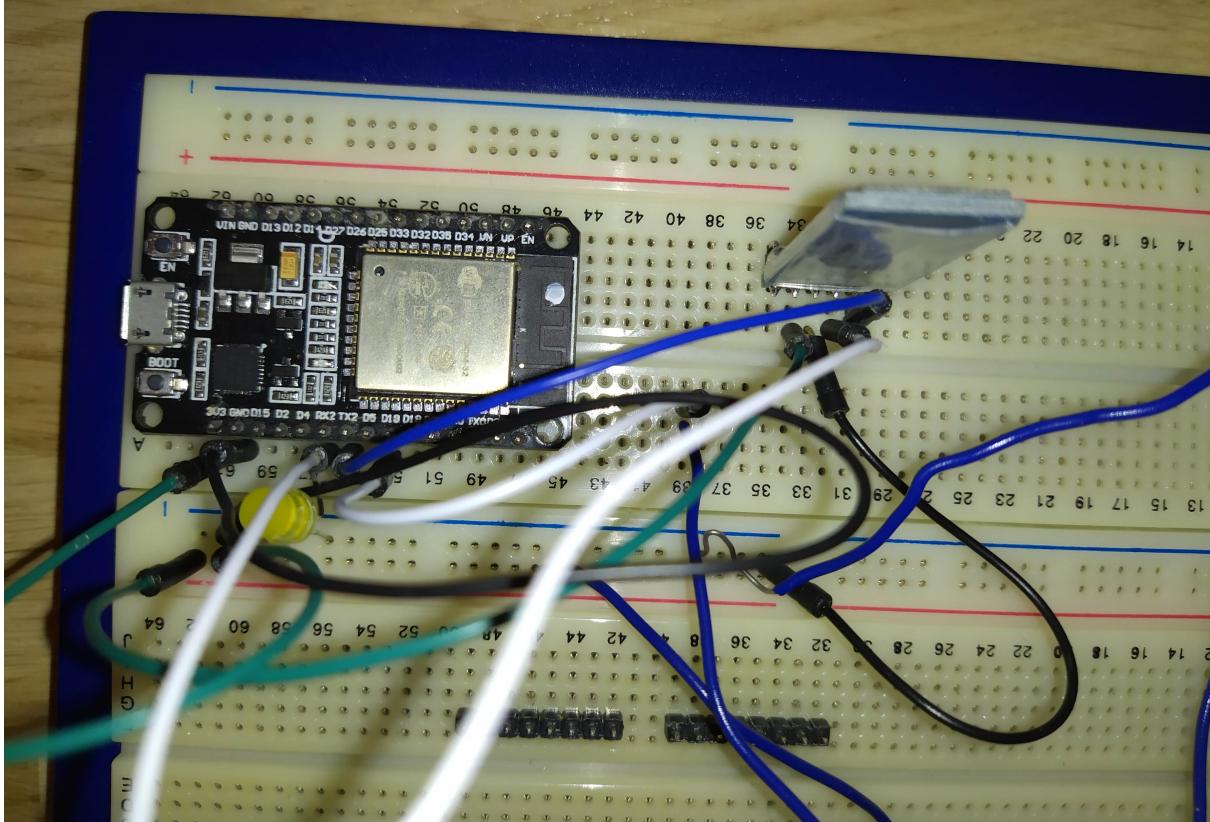
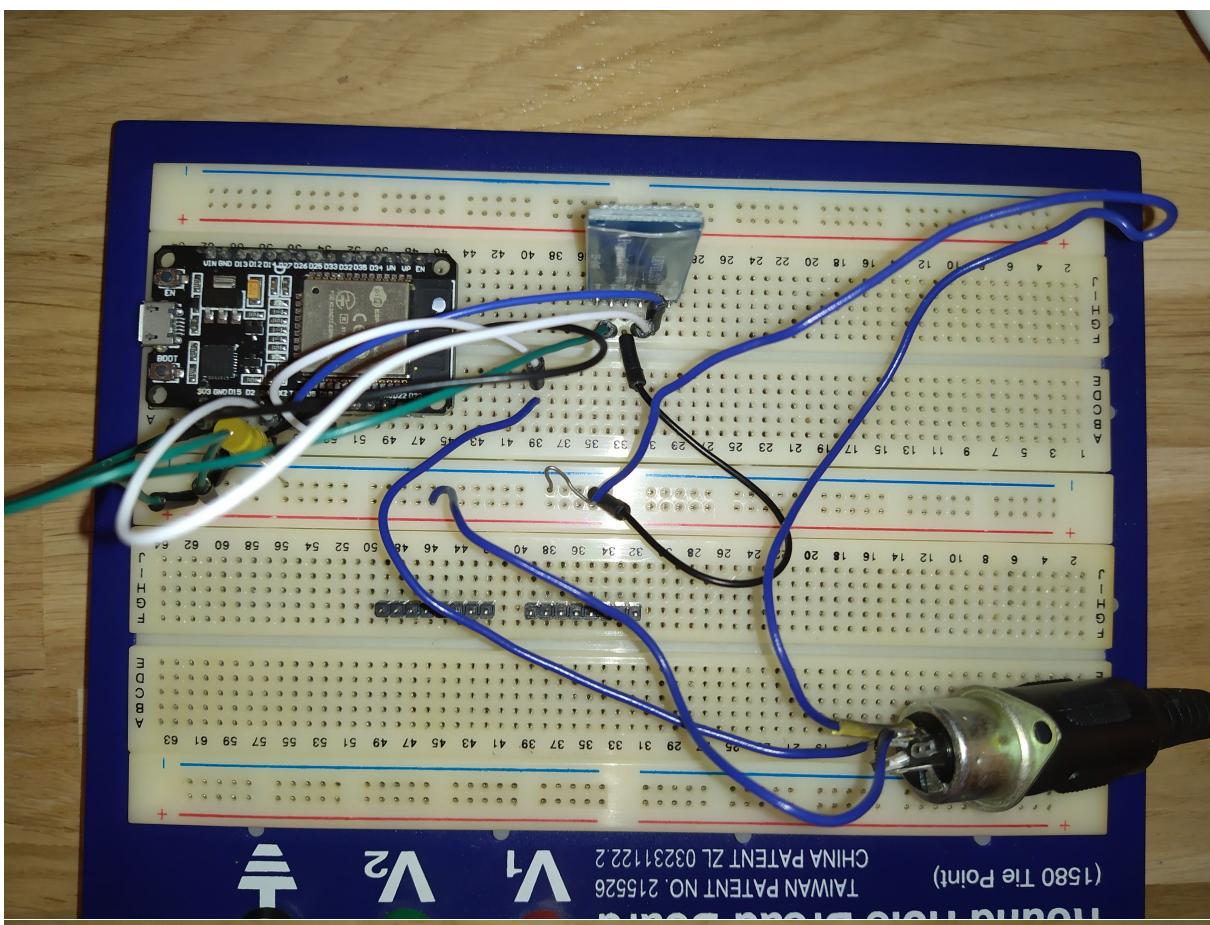
https://youtu.be/_9DkzAXo7i8

Na tym filmie odbiornik jest podłączony przez złącze MIDI do wzmacniacza Kemper Profiling Amplifier. Poprzez komunikaty typu *Control Change* są zdalnie prełączane presety:

<https://youtu.be/wr5JzCO7T9I>

Poniżej są przedstawione zdjęcia nadajnika i odbiornika:





Literatura

- [1] Dokumentacja języka MicroPython
- [2] Dokumentacja biblioteki SoftwareSerial
- [3] Poradnik do debounce-owania przycisków na Arduino
- [4] Poradnik do tworzenia własnych bibliotek Arduino
- [5] Pinout płyty ESP-32
- [6] Pinout i specyfikacja ATTINY85
- [7] Specyfikacja komunikatów MIDI
- [8] Specyfikacja złącza DIN-5 dla MIDI
- [9] Poradnik do programowania ATTINY85 przez Arduino Nano
- [10] Dokumentacja frameworku Kivy
- [11] Dokumentacja frameworku MicroWebSrv
- [12] Komendy AT dla modułu HM-10 MLT-BT05