Telefon - dokumentacja końcowa

Zespół

- Damian Kolaska
- Kamil Przybyła
- Michał Szaknis

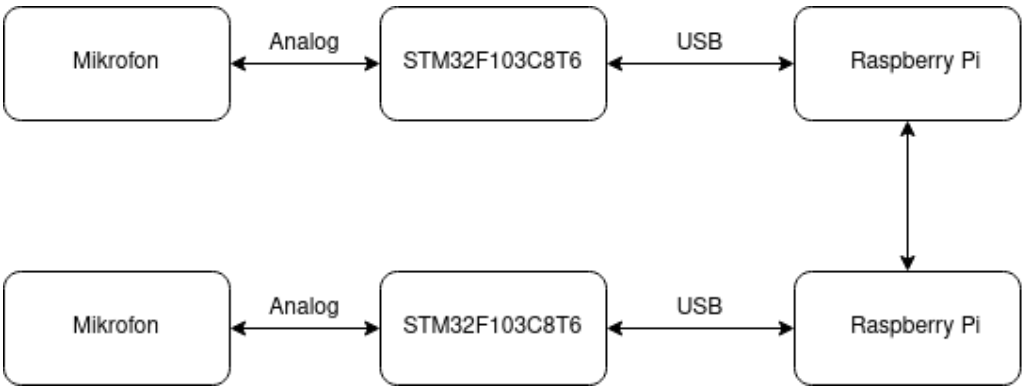
Repozytorium

https://gitlab-stud.elka.pw.edu.pl/deratyzacja/telefon

Cel projektu

Celem projektu była realizacja systemu umożliwiającego dwukierunkową transmisję sygnału z mikrofonu. Wykorzystano dwa minikomputery Raspberry Pi oraz mikrokontrolery STM32. Rolą mikrokontrolerów jest konwersja danych analogowych, pochodzących z mikrofonu do sygnału cyfrowego, który zostanie przetworzony przez proces działający na Raspberry Pi.

Schemat ogólny

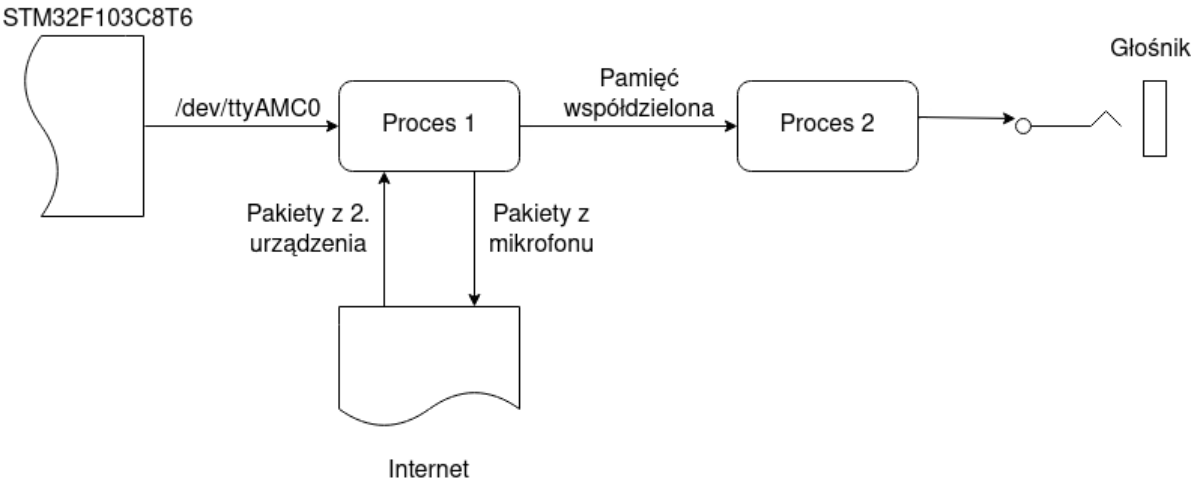


Sprzęt

- Raspberry Pi 3B+
- Raspberry Pi 4
- Moduł mikrofonowy ze wzmacniaczem MAX9814

- Mikrokontroler STM32F103C8T6

Komunikacja



Mikrofon <-> STM32F103C8T6

Komunikacja pomiędzy mikrofonem a mikrokontrolerem odbywa się za pośrednictwem kanału analogowego. Na wyjściu mikrofonu otrzymujemy sygnał postaci:

$$V(t) = V_0 + v(t)$$

V_0 - składowa stała ok. $1.25V$
$v(t)$ - składowa zmienna $-1 \leq v(t) \leq 1$

STM32F103C8T6 <-> Raspberry Pi

Ze względu na ilość przesyłanych danych zdecydowaliśmy się na wykorzystanie wbudowanego w mikrokontroler interfesju USB do dalszej komunikacji. Maksymalna prędkość transmisji wbudowanego interfejsu wynosi ok. $12MB/s$, co zapewnia duży margines na ewentualne zwiększenie częstotliwości próbkowania przetwornika ADC wbudowanego w mikrokontroler. Ustaliliśmy częstotliwość próbkowania na poziomie $20KHz$.

Raspberry Pi <-> Raspberry Pi

Komunikacja pomiędzy urządzeniami końcowymi przebiega za pomocą łącza internetowego. Protokołem warstwy transportowej jest protokół TCP.

Procesy pracujące na Raspberry Pi

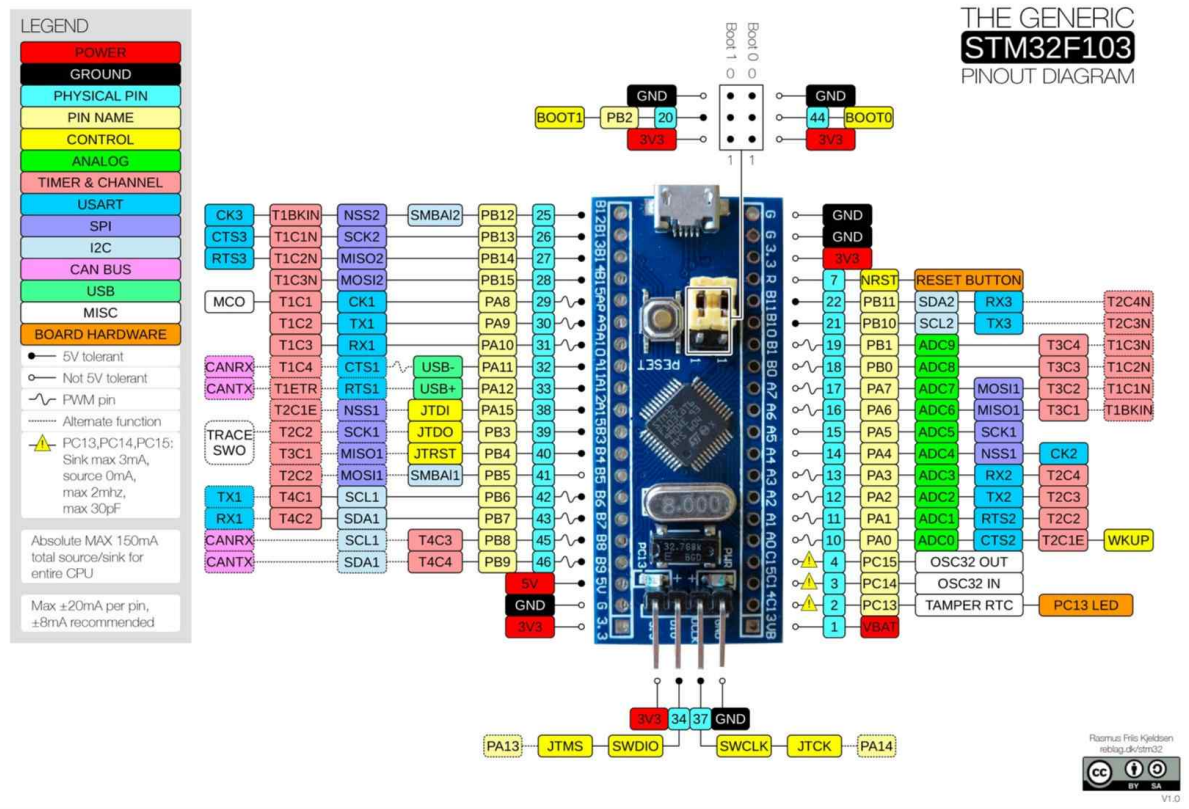
Proces 1

Zadaniem procesu 1. jest odebranie danych z mikrofonu, wysłanie ich przez sieć, równoczesne odbieranie pakietów z sieci i umieszczenie ich w pamięci współdzielonej tak, aby proces 2. mógł z nich skorzystać.

Proces 2

Proces 2. odpowiada za przyjmowanie danych z współdzielonego bufora, w którym znajdują się pakiety próbek dźwięku oraz odtwarzanie tego dźwięku, przy wykorzystaniu biblioteki *RtAudio*.

Oprogramowanie mikrokontrolera



Zadaniem mikrokontrolera jest konwersja analogowego sygnału z mikrofonu na cyfrowy, korzystając z wbudowanego przetwornika analogowo-cyfrowego pracującego na częstotliwości $12MHz$ co pozwala na na uzyskanie czasu próbkowania na poziomie ok. $2\mu s$. Drugim ważnym zadaniem jest wstępne buforowanie odczytanych wartości w pakiety po 1024 rekordy w celu relaksacji wymagań pracy w czasie rzeczywistym (dzięki temu na Raspberry Pi nałożone są wymagania czasowe na poziomie kilku milisekund a nie dziesiątek mikrosekund).

Oprogramowanie Raspberry Pi

Minikomputery Raspberry Pi pracują pod systemem Debian w wersji 64-bitowej oraz Raspbian w wersji 32-bitowej.

Sposób uruchomienia

Uruchomienie systemu polega na włączeniu na maszynie Raspberry Pi procesu pierwszego, podając odpowiednie argumenty wywołania:

```
./proces1 -host|-client ip port [-s|-r]
```

gdzie flagi `-host` oraz `-client` są wzajemnie wykluczające się i oznaczają rolę, jaką ma pełnić maszyna, `ip` oznacza adres maszyny do której się łączymy, `port` to port na którym przebiega komunikacja, a opcjonalne flagi `-s` oraz `-r` służą do ograniczenia komunikacji w jedną stronę (odpowiednio: tylko wysyłanie danych i tylko ich odbiór).

Konieczne jest również uruchomienie procesu drugiego, odpowiedzialnego za odtwarzanie dźwięku:

```
./proces2
```

Parametry systemu

Wszystkie stałe wykorzystywane przez system, zdefiniowane są we wspólnym pliku nagłówkowym `config.hpp`:

```
const unsigned int DEQUE_SIZE = 1024;

const unsigned int BUFFER_SIZE = 1024;
const uint32_t PACKET_SIZE = BUFFER_SIZE * 2 + 8;
const std::size_t DATA_SIZE = PACKET_SIZE - 8;

const uint16_t constant_compound = 1551; // (1.25/3.3)*4096

const char SHM_AUDIO_TEST_NAME[] = "/test_audio";
const unsigned int NUM_FRAMES = 5012;
const unsigned int SAMPLING_RATE = 20000;
```

- *DEQUE_SIZE* - rozmiar kolejki przechowującej pakiety audio
- *BUFFER_SIZE* - liczba próbek w pakiecie audio
- *PACKET_SIZE* - rozmiar struktury pakietu wysyłanego przez sieć (w bajtach)
- *DATA_SIZE* - rozmiar tablicy próbek audio w bajtach
- *constant_compound* - składowa stała sygnału odbieranego z mikrofonu
- *SHM_AUDIO_TEST_NAME* - nazwa pliku pamięci współdzielonej
- *NUM_FRAMES* - liczba próbek w pakiecie (stała używana tylko w testach)
- *SAMPLING_RATE* - częstotliwość próbkowania

API

Komunikacja między procesami

Pamięć współdzielona

Klasa będąca realizacją ideologii RAII. Pośredniczy przy dostępie do pamięci współdzielonej i gwarantuje jej zwolnienie, po tym jak wszystkie instancje tej klasy zostaną usunięte.

```
template<typename T>
class shared_mem_ptr {

    struct ref {
        std::size_t use_count = 0;
        T obj;
    };

    ref* ptr;
    int memfd;
    const char* path;
public:

    template<typename ... Args>
    shared_mem_ptr(const char* ptr, Args&&... args);
    shared_mem_ptr(const shared_mem_ptr& other);
    shared_mem_ptr(shared_mem_ptr&& other);

    ~shared_mem_ptr();

    T* get() noexcept { return &ptr->obj; }
```

```
    const T* get() const noexcept { return &ptr->obj; }
    T& operator*() noexcept { return ptr->obj; }
    const T& operator*() const noexcept { return ptr->obj; }
    T* operator->() noexcept { return &ptr->obj; }
    const T* operator->() const noexcept { return &ptr->obj; }
    shared_mem_ptr<T>& operator=(const shared_mem_ptr& other);
    shared_mem_ptr<T>& operator=(shared_mem_ptr&& other);
};
```

Kolejka pakietów

Bufor cykliczny, wykorzystywany do przechowywania pakietów próbek dźwiękowych.

```
template<typename T, std::size_t N>
class fast_deque {
    static constexpr std::size_t invalid_index =
std::numeric_limits<std::size_t>::max();

    std::array<T, N> array;
    std::size_t front_index = invalid_index;
    std::size_t back_index = 0;

    static std::size_t next(std::size_t n) noexcept;
    static std::size_t prev(std::size_t n) noexcept;
public:
    typename std::array<T, N>::iterator push_front() noexcept;
    typename std::array<T, N>::iterator pop_front() noexcept;

    typename std::array<T, N>::iterator push_back() noexcept;
    typename std::array<T, N>::iterator pop_back() noexcept;

    typename std::array<T, N>::iterator frontit() noexcept { return
std::next(array.begin(), front_index); };
    typename std::array<T, N>::iterator backit() noexcept { return
std::next(array.begin(), back_index); };

    T& front() noexcept { return array[front_index]; }
    const T& front() const noexcept { return array[front_index]; }

    T& back() noexcept { return array[back_index]; }
    const T& back() const noexcept { return array[back_index]; }

    bool valid(typename std::array<T, N>::iterator it) const noexcept { return it
!= array.end(); }
    bool empty() const noexcept;
    bool full() const noexcept;
    void reset() noexcept;

    typedef typename std::array<T, N>::iterator iterator;
};
```

Spin lock

Synchronizuje dostęp do pamięci.

```
template<typename T>
class spin_locked_resource {
    std::atomic_bool lock_;
    T obj;

    class locked_resource {
        spin_locked_resource& lock;
        T& obj;
    public:
        locked_resource(spin_locked_resource& lock, T& obj);
        locked_resource(const locked_resource& ) = delete;
        locked_resource(locked_resource&& ) = delete;
        ~locked_resource();

        T* operator->() noexcept { return &obj; }
        T& operator*() noexcept { return obj; }

        auto operator=(const locked_resource& ) = delete;
    };

    void unlock() noexcept;
public:
    template<typename ... Args>
    spin_locked_resource(Args&&... args);

    locked_resource lock() noexcept;
};
```

Semafor

```
class Semaphore {
    int id;
    static int next_proj_id();
public:
    Semaphore(int value = 1);
    Semaphore(const char* path, int proj_id, int value = 1);
    Semaphore(const Semaphore& other);
    void P();
    void V();
};
```

Audio

Klasa, będąca interfejsem do biblioteki *RtAudio*. Służy do odtwarzania dźwięku, którego próbki będą zawarte w buforze zapełnianym w czasie rzeczywistym.

```
template <unsigned int frames = 64u>
class Audio {
public:
  struct AudioPacket {
    uint16_t data[frames];
    uint32_t reserved;
```

```
  };

  using PacketDeque = shared_mem_ptr<spin_locked_resource<fast_deque<AudioPacket,
DEQUE_SIZE>>>;
private:
  RtAudio dac_;
  RtAudio::StreamParameters params_;
  unsigned int sampleRate_;
  unsigned int bufferFrames_;
  std::optional<PacketDeque> buffer_;
public:
  Audio(unsigned int sampleRate);
  ~Audio();
  void play(PacketDeque ptr);
  void stop();

  unsigned int getSampleRate() const { return sampleRate_; }
  unsigned int getBufferFrames() const { return bufferFrames_; }

  static int transfer(void* outputBuffer, void* inputBuffer, unsigned int
nBufferFrames, double streamTime, RtAudioStreamStatus status, void* userData);
};
```

Komunikacja sieciowa

W związku z problemami wynikającymi z usług oferowanych przez naszych dostawców internetu, byliśmy zmuszeni do wykorzystania protokołu TCP.

Protokół sieciowy

Protokół sieciowy zaimplementowaliśmy z wykorzystaniem biblioteki *Google Protocol Buffers*. Nasz wybór został podyktowany faktem niezależności stworzonego w niej protokołu od architektury procesora danej platformy.

```
syntax = "proto3";

message Data {
    string data = 1;
}

message Request {
    oneof Content {
        Data data = 2;
    }
}

message Response {
    oneof Content {
        Data data = 2;
    }
}
```

Aplikacje testowe

Poniżej znajduje się lista aplikacji testowych, znajdujących się w katalogu `tests`:

- *ipc* - służy do weryfikacji poprawności działania pamięci współdzielonej
- *deque_test* - aplikacja testująca działanie bufora cyklicznego
- *sem* - badanie poprawności działania semafora
- *spin_lock* - test spin locka
- *network_tcp* - wysyła i odbiera pakiety TCP, imitując działanie programu ping
- *network_udp* - analogiczne działanie, tym razem z pakietami UDP
- *mic_dump_to_file* - odczytywanie pakietów audio z mikrofonu i zapisywanie ich do pliku
- *audio_generator* - generuje pakiety sygnału sinusoidy i wrzuca je do pamięci współdzielonej
- *audio_player* - odczytuje i odtwarza pakiety audio, znajdujące się w pamięci współdzielonej
- *mock_mic_generator* - tworzy nazwaną kolejkę, która ma imitować mikrofon; generuje sygnał sinusoidalny (używana do testowania systemu przez osoby niemające dostępu do sprzętu)
- *audio_from_mic* - tworzy proces pobierający dane z mikrofonu i odtwarzający je
- *audio_form_mic_over_network* - pobieranie danych z mikrofonu oraz sieci, wysyłanie pakietów oraz odtwarzanie pakietów odebranego dźwięku

Metoda pomiaru opóźnień

Z powodu braku dokładnego timera w STM'ie opóźnienie mierzymy od momentu wysłania pakietu w STM'ie do otrzymania go w procesie 2. (wyjęciu procesu z kolejki).
Do pomiaru opóźnień wykorzystujemy dodatkowe połączenie między mikrokontrolerem a Raspberry Pi

Raspberry Pi pin	**stm32 pin**	**Opis**
GPIO 8	GPIOA_5	Pin Enable / Disable służy do wstrzymania / kontynuowania wysyłania pakietów
GPIO 9	GPIOA_4	Mikrokontroler zmienia stan pinu w momencie wysłania pakietu do peryferium USB

Na początku program benchmarkujący wyłącza wysyłanie pakietów i czeka ok. 1s w celu upewnienia się, że wszystkie pakiety, które były wysłane dotarły już do Raspberry Pi. Następnie czyści kolejkę pakietów i gdy jest pusta odblokowuje wysyłanie pakietów. Następnie w pętli sprawdzane są 2 rzeczy:

1. Czy do kolejki został dodany nowy pakiet - wtedy zapisujemy timestamp odebrania
2. Czy stan pinu GPIO 9 się zmienił wtedy zapisujemy timestamp wysłania kolejnego pakietu

W celu rozróżniania pakietów wprowadziliśmy dodatkowe pole będące numerem pakietu. Numery są przyznawane od 0 i licznik jest resetowany gdy GPIOA_5 == 0. Pozwala to na identyfikacje pakietu z jego timestampem wysłania jak i detekcje liczby pakietów traconych. W celu zapewnienia większej dokładności wątek badający stan pinu został przypisany do rdzenia odizolowanego. Dzięki temu mamy pewność że zawsze zarejestrujemy czas wysłania z największą możliwą dokładnością. Natomiast wątek sprawdzający czy do kolejki dodany został nowy pakiet pozostał na normalnym wątku w celu umożliwienia pomiaru opóźnień spowodowanych planistą systemowym.

Wyniki eksperymentów

Eksperymenty wykonywane na Raspberry Pi 3B+

Rozmiar pakietu (liczba zawartych próbek) ma bardzo duże znaczenie na jakość komunikacji. Zmieniając parametry zauważyliśmy iż pakiet posiadający 128 próbek działa dobrze tylko na komputerze osobistym (z procesorem o taktowaniu ~$4GHz$). Przy próbie na Raspberry Pi słychać było znaczące przerwy w odtwarzaniu powodowane mniejszą responsywnością systemu. Aby temu zaradzić zwiększaliśmy wielkość pakietu. Pakiet wielkości 256 działał już dobrze (ale bez sieci, z przesyłaniem bezpośrednio do głośnika). Niestety przy próbie komunikacji po sieci okazało się, że opóźnienia sieciowe uniemożliwiają transmisje, zwiększyliśmy wtedy wielkość pakietu do 1024 próbek co pozwoliło na swobodną komunikację dwustronną.

Przy pakiecie o wielkości 1024 oraz $Fp = 20kHz$ wysyłamy w każdym pakiecie fragment transmisji o długości $1024 * (1/Fp) = 51.2ms$ co pozwala na transmisje po łączach z o średnich opóźnieniach mniejszych od ~$50ms$.

Czas od wysłania pakietu z STM do odtworzenia go przez RTAudio (bez sieci)

Raspberry Pi 3B+

liczba próbek	**128**	**256**	**512**	**1024**
opóźnienie	---	1.86 ms	2.85 ms	4.87 ms

Największym problemem okazały się stałe opóźnienia związane z przesyłaniem danych do karty dźwiękowej, które powodowały, że dla małych pakietów które pokrywały mniej osi czasu niż wynosiły sumaryczne opóźnienia (np. pakiet 256 pokrywa ok. 12.8ms)

Rezerwacja rdzeni dla procesów

Zarezerwowanie rdzeni 2 i 3 poprzez modyfikację pliku `/boot/cmdline.txt`:

```
... isolcpus=2,3
```

Przypisanie procesów do rdzeni:

```
cpu_set_t set;
CPU_ZERO(&set);
CPU_SET(cpu_num, &set); // 2 lub 3 w zależności od procesu
sched_setaffinity(getpid(), sizeof(set), &set);
```

Zajęcie pozostałych rdzeni poleceniem stress:

```
stress -c 2
```

Wynik:

```
0 packets was lost and mean delay was 4884.243243 microseconds
```

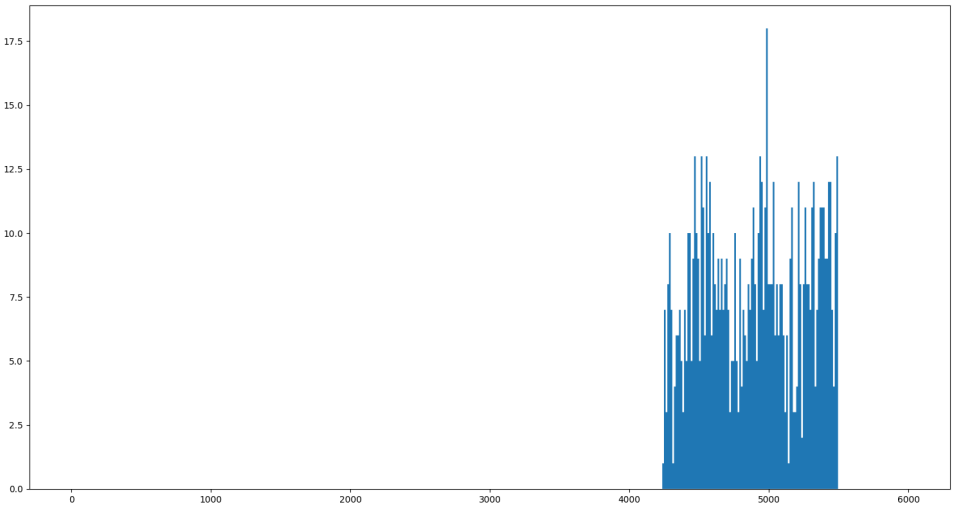
Wynik bez rezerwacji:

```
0 packets was lost and mean delay was 4890.920000 microseconds
```

Wniosek:
Izolowanie rdzeni dla procesów czasu rzeczywistego nie ma istotnego wpływu na średni czas odpowiedzi w przypadku jeśli do dyspozycji pozostały jeszcze nieobciążone rdzenie procesora.

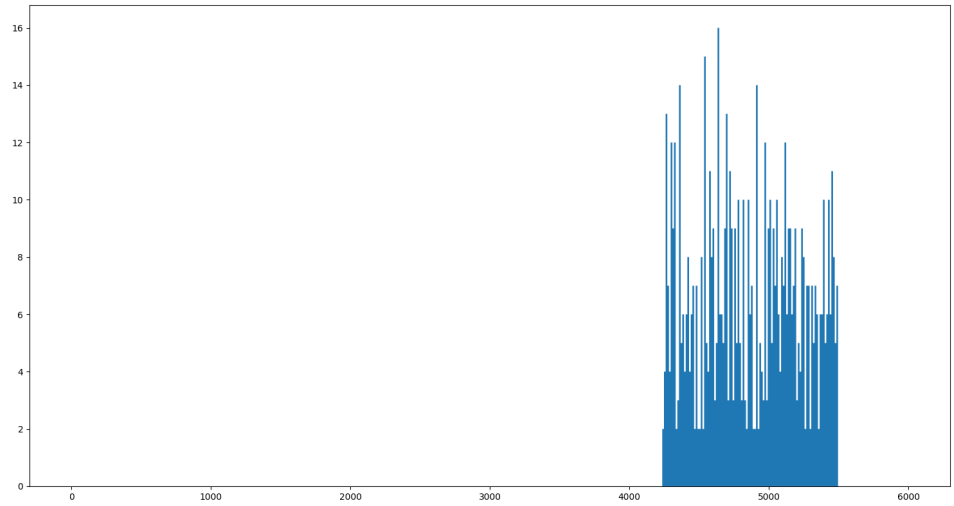
Czasy wykonania dla różnych scenariuszy wykonania

Bez rezerwacji rdzeni, bez obciążenia



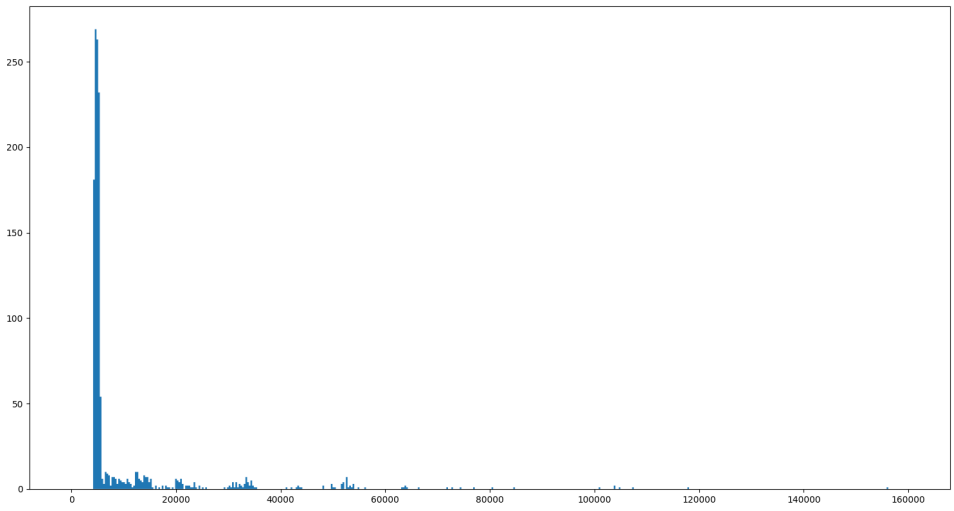

```
0 packets was lost and mean delay was 4896.258929 microseconds, standard
deviation: 459.656680
Maximum delay: 7861.000000
```

Bez rezerwacji rdzeni, umiarkowane obciążenie (2 rdzenie)



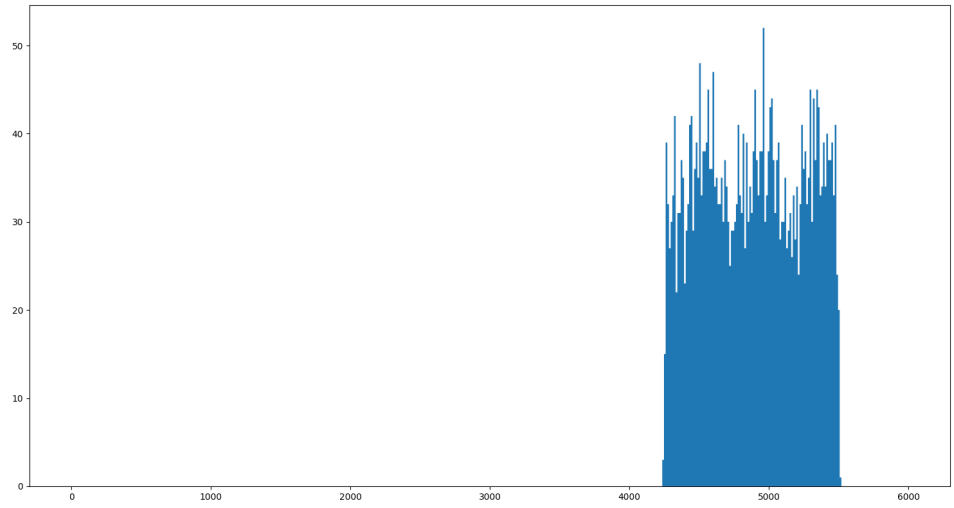

```
0 packets was lost and mean delay was 4864.831897 microseconds, standard
deviation: 363.676840
Maximum delay: 5495.000000
```

Bez rezerwacji rdzeni, maksymalne obciążenie (3 rdzenie)




```
1 packets was lost and mean delay was 9629.374526 microseconds, standard
deviation: 13510.689862
Maximum delay: 156018.000000
```

Rezerwacja rdzeni 2 i 3, maksymalne obciążenie




```
0 packets was lost and mean delay was 4883.412834 microseconds, standard
deviation: 360.493927
Maximum delay: 5512.000000
```

Na pierwszych wykresach widać nagłe odcięcie wartości krańcowych, które miało wpływ na rozkład opóźnień. Jednakże z wykresu 3. widać iż występuje tzw. "ogon". Co oznacza, iż gdybyśmy zebrali więcej próbek to istnieje możliwość że pozostałe wykresy rozszerzyłyby się o krańcowe próbki.

Uwagi i wnioski

Udało się nam dotrzymać wymogów pracy w trybie czasu rzeczywistego bez modyfikacji bazowego kernela. Było to możliwe, gdyż nasz komputer nie był obciążony żadnym innym zadaniem, które zajmowałby więcej niż 2 rdzenia procesora. Przeprowadzając eksperymenty zaobserwowaliśmy, iż przy obciążeniu więcej niż 2 rdzeni (np. programem stress) opóźnienia pracy naszego systemu zaczynają bardzo szybko wzrastać - z początku nawet o rzędy wielkości. Wtedy konieczne staje się przypisanie procesów do rdzeni i odizolowanie ich od planisty systemowego.

Podczas prac okazało się że:

- komunikacja przez UART jest za wolna już przy bitrate'cie ok. 1MB/s
- Linuksowy sterownik pseudoterminali, bez ustawienia olbrzymiej liczby opcji, dotyczących parametrów transmisji (cfmakeraw itd.), powodował iż część bajtów z pakietu nie dochodziła do programu odczytującego

Podział prac

- Damian Kolaska - komunikacja sieciowa, odbieranie, walidacja i pakietowanie danych z mikrokontrolera, testy funkcjonalne, przeprowadzenie testów opóźnień.
- Kamil Przybyła - warstwa abstrakcji nad *RtAudio*, przekazywanie pakietów audio z bufora współdzielonego, wykorzystanie biblioteki Protobuf
- Michał Szaknis - wybór sprzętu, zaprogramowanie mikrokontrolera, projekt protokołów komunikacyjnych, API do komunikacji międzyprocesowej, napisanie testu opóźnień.

Wyniesiona wiedza

- Damian Kolaska - komunikacja sieciowa przy użyciu gniazd sieciowych, obsługa deskryptorów plików, podstawowy obsługi STM Cube IDE oraz programatora ST LINK v2, szukanie błędów buffer overflow przy użyciu ASAN
- Kamil Przybyła - podstawowe użycie biblioteki Protobuf, działanie pamięci współdzielonej, poznanie biblioteki RtAudio
- Michał Szaknis - driver do pseudoterminali potrafi zfrustrować poprzez escapowanie bajtów, obłsuga USB na stm32