5월 1주차 보고서 (2024.04.29 ~ 2024.05.05)

💡 2024.04.29(월) 11:00 ~ 18:00 / 오프라인 회의 / 참여자: 김지윤, 박수진, 이다은

💡 2024.04.30(화) 11:00 ~ 16:00 / 오프라인 회의 / 참여자: 김지윤, 박수진, 이다은

💡 2024.05.01(수) 19:30 ~ 22:00 / 오프라인 회의 / 참여자: 김지윤, 박수진, 이다은, 정지원

💡 2024.05.02(목) 11:00 ~ 15:00
참여자: 박수진, 이다은
오프라인 개발

💡 2024.05.02(목) 13:00 ~ 21:00
참여자: 김지윤, 정지원
오프라인 개발

💡 2024.05.03(금) 11:00 ~ 15:00
참여자: 박수진, 이다은
오프라인 개발

💡 2024.05.03(금) 11:00 ~ 17:00
참여자: 김지윤, 정지원
오프라인 개발

💡 2024.05.05(일) 11:30 ~ 18:00
참여자: 박수진, 이다은
오프라인 개발

💡 2024.05.03(금) 21:00 ~ (토)01:00
참여자: 김지윤
오프라인 개발

<소프트웨어>

<소프트웨어>
- USB Audio Device Port확인
- PiDTLN(ns.py)
  - ns.py 실행 과정
  - 오디오 장치 정보 조회 및 실시간 소리 출력 테스트
  - 샘플링 레이트 변환 방법
  - ns_0429.py
  - ns_0430.py
  - ns_0501.py
  - *TODO*
- 실시간 노이즈 제거 및 실시간 오디오 출력
  - RealTime_Noise.py on RasberryPi
  - 두 개의 실시간 입출력
  - 10초간 두 개의 실시간 입출력을 통해 각 RMS 및 평균 RMS 구하기
  - 10초간 두 개의 실시간 입출력을 통해 각 RMS 및 평균 RMS 구한 후, 실시간 출력을 평균 RMS에 맞춰서 출력하기

<하드웨어>
- GPIO 제어
  - **1. 릴레이 모듈 설정**
  - **2. 15dB 증폭을 위해 릴레이 제어**
  - **3. 마이크로 오디오 입력 수신 및 I2S를 통한 전송**
- 추가 이슈
- 다음 할 일
  - **I2S 활성화**
  - **라즈베리파이 5 I2S 핀 연결**

실시간 data 전송
- **1. 릴레이 모듈 제어 코드**
- **2. 마이크로부터 실시간 오디오 수신**
- **3. I2S를 통한 오디오 전송 코드**
- **전체 통합**
- GPIO 제어
- 다음 할 일
- 릴레이제어 - LED
- 4채널 릴레이 두 개 제어
- 라즙베리파이 - 증폭모듈연결
- 마이크 동시 입력
- <최종 확인>

다음 할 일

USB Audio Device Port확인

- 입력장치

```
arecord -l

**** List of CAPTURE Hardware Devices ****

# ... 생략

card 2: Device [USB Audio Device], device 0: USB Audio [USB A
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

- 출력장지

```
aplay -l

**** List of PLAYBACK Hardware Devices ****

# ... 생략

card 2: Device [USB Audio Device], device 0: USB Audio [USB A
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

- devices.py

```
import sounddevice as sd
print(sd.query_devices())
```

```
python devices.py

     # ... 생략

   7 HDA Intel PCH: HDMI 4 (hw:1,10), ALSA (0 in, 8 out)
```

```
    8 USB Audio Device: - (hw:2,0), ALSA (1 in, 2 out)
    9 sysdefault, ALSA (128 in, 128 out)
   10 front, ALSA (0 in, 32 out)
   11 surround21, ALSA (32 in, 32 out)

   # ... 생략

 * 23 default, ALSA (32 in, 32 out)
```

PiDTLN(ns.py)

1. **오디오 장치 설정과 스트림 처리**
    - 사용자는 커맨드 라인 인자를 통해 입력(i) 및 출력(o) 오디오 장치 지정 → 이 장치들은 샘플링 레이트가 16000Hz로 설정되어 있어야 한다.
    - sounddevice.Stream을 사용하여 실시간으로 오디오 스트림 처리 → 이 스트림은 입력 오디오를 받아 모델을 통해 처리한 후 출력 오디오로 전송
2. **디지털 신호 처리**
    - 입력된 오디오 데이터는 블록 단위로 처리 → 각 블록은 32ms 길이로 설정되며, 8ms 마다 새로운 오디오 데이터로 업데이트
    - FFT(Fast Fourier Transform)를 사용하여 오디오 데이터의 주파수 영역 표현을 계산 → 이를 통해 노이즈 서프레션 모델이 주파수 영역에서 작동할 수 있게 한다.
3. **모델 처리**
    - TensorFlow Lite 모델(tflite.Interpreter)을 사용하여 노이즈를 감소시키는 마스크 생성
    - 첫 번째 모델은 입력된 주파수 데이터에 대해 마스크 계산, 두 번째 모델은 이 마스크를 적용한 후 시간 영역으로 데이터 복원
4. **출력**
    - 처리된 오디오 데이터는 다시 시간 영역으로 변환되어 출력 버퍼에 저장된 후, 지정된 출력 장치로 전송

주의점:

- 이 스크립트는 입력과 출력 장치가 모두 *16000Hz의 샘플링 레이트를 지원하도록 설정되어야 제대로 작동*한다.

- 만약 다른 샘플링 레이트의 오디오 장치를 사용하려면, *오디오 샘플링 레이트를 변환하는 추가적인 처리 로직 필요*

ns.py 실행 과정

```
python ns_daeun.py -i 8 -o 8 --measure

INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
Expression 'paInvalidSampleRate' failed in 'src/hostapi/alsa/
Expression 'PaAlsaStreamComponent_InitialConfigure( &self->ca
Expression 'PaAlsaStream_Configure( stream, inputParameters,
PortAudioError: Error opening Stream: Invalid sample rate [Pa
```

- PortAudio 라이브러리를 사용하여 오디오 스트림을 열려고 할 때 발생
- 주로 오디오 장치에서 지원하지 않는 샘플 레이트(sample rate) 사용 시도 시 나타남
- 오류 코드 -9997은 paInvalidSampleRate 즉, 유효하지 않은 샘플 레이트를 나타냄

1. **지원되는 샘플 레이트 확인**: 사용 중인 오디오 장치가 지원하는 샘플 레이트를 확인 → 일반적으로 오디오 장비는 표준 샘플 레이트(예: 44100 Hz, 48000 Hz 등)를 지원

```
import sounddevice as sd
print(sd.query_devices(8))
```

```
python devices.py

{'name': 'USB Audio Device: - (hw:2,0)', 'index': 8, 'host
'max_input_channels': 0, 'max_output_channels': 2,
'default_low_input_latency': -1.0,
'default_low_output_latency': 0.008707482993197279,
'default_high_input_latency': -1.0,
'default_high_output_latency': 0.034829931972789115,
'default_samplerate': 44100.0}
```

2. **PortAudio 설정 검토**: 오디오 스트림을 설정할 때 적절한 입력 및 출력 파라미터가 사용되었는지 확인 → 코드에 설정된 샘플 레이트, 프레임 버퍼 크기 등이 장치와 호환되는지 검토

   1) 파라미터 로깅

   스크립트에서 오디오 스트림을 설정하는 부분에 로깅을 추가하여, 실제로 사용된 입력 및 출력 파라미터 값을 확인 → Python에서 sounddevice 모듈을 사용하여 스트림을 열 때 사용된 파라미터 출력

```
pythonCopy code
import sounddevice as sd

# 오디오 스트림 설정 예
def open_stream(input_device, output_device, fs, blocksi
ze, channels):
    try:
        with sd.Stream(device=(input_device, output_devi
ce),
                       samplerate=fs,
                       blocksize=blocksize,
                       dtype='float32',
                       channels=channels,
                       callback=callback) as stream:
            print(f"Stream opened with parameters: {stre
am}")
            print(f"Input Device: {input_device}")
            print(f"Output Device: {output_device}")
            print(f"Sample Rate: {fs}")
            print(f"Block Size: {blocksize}")
            print(f"Channels: {channels}")
            # 스트림 처리 코드
            ...
    except Exception as e:
        print("Error opening stream: ", e)

# 스트림 열기
open_stream(input_device=8, output_device=8, fs=44100, b
locksize=1024, channels=(1, 1))
```

2) 오디오 장치 정보 검증

스크립트 실행 전에 해당 오디오 장치가 스트림 설정에 필요한 샘플 레이트와 채널 구성을 지원하는지 확인 → sounddevice.query_devices() 함수를 사용하여 오디오 장치의 지원 정보확인

```
pythonCopy code
def check_device_capabilities(device_index):
    device_info = sd.query_devices(device_index, 'inpu
t')
    print(f"Device {device_index} capabilities:")
    print(f"  Max Input Channels: {device_info['max_inpu
t_channels']}")
    print(f"  Max Output Channels: {device_info['max_out
put_channels']}")
    print(f"  Default Sample Rate: {device_info['default
_samplerate']}")

check_device_capabilities(8)
```

오디오 장치 정보 조회 및 실시간 소리 출력 테스트

- check.py

```
import sounddevice as sd
print(sd.query_devices(8))

import sounddevice as sd

def callback(indata, outdata, frames, time, status):
    if status:
        print(status)
    outdata[:] = indata

def open_stream(input_device, output_device, fs, blocksize, c
    try:
```

```
        with sd.Stream(device=(input_device, output_device),
                       samplerate=fs,
                       blocksize=blocksize,
                       dtype='float32',
                       channels=channels,
                       callback=callback) as stream:
            print(f"Stream opened with parameters: {stream}")
            print(f"Input Device: {input_device}")
            print(f"Output Device: {output_device}")
            print(f"Sample Rate: {fs}")
            print(f"Block Size: {blocksize}")
            print(f"Channels: {chaㅁnnels}")
            while True:
                sd.sleep(1000)  # 스트림 유지
    except Exception as e:
        print("Error opening stream: ", e)

def check_device_capabilities(device_index):
    device_info = sd.query_devices(device_index)
    print(f"Device {device_index} capabilities:")
    print(f"  Max Input Channels: {device_info['max_input_cha
    print(f"  Max Output Channels: {device_info['max_output_c
    print(f"  Default Sample Rate: {device_info['default_samp

# 사용 예
check_device_capabilities(8)
open_stream(input_device=8, output_device=8, fs=44100, blocks
```

```
{'name': 'USB Audio Device: - (hw:2,0)', 'index': 8, 'hostapi
'max_input_channels': 1, 'max_output_channels': 2,
'default_low_input_latency': 0.008707482993197279,
'default_low_output_latency': 0.008707482993197279,
'default_high_input_latency': 0.034829931972789115,
'default_high_output_latency': 0.034829931972789115,
'default_samplerate': 44100.0}

Device 8 capabilities:
  Max Input Channels: 1
```

```
  Max Output Channels: 2
  Default Sample Rate: 44100.0
Stream opened with parameters: <sounddevice.Stream object at 
Input Device: 8
Output Device: 8
Sample Rate: 44100
Block Size: 16384
Channels: (1, 2)
```

샘플링 레이트 변환 방법

샘플링 레이트 변환: 입력된 오디오를 16000Hz로 다운샘플링하고, 처리가 끝난 후 다시 44100Hz로 업샘플링하는 과정

```
pip install librosa
```

```
import librosa

# 샘플링 레이트 변환 함수
def resample_audio(data, original_sr, target_sr):
    return librosa.resample(data, orig_sr=original_sr, targ
et_sr=target_sr)

# 콜백 함수 내에서 사용
def callback(indata, outdata, frames, time, status):
    global in_buffer, out_buffer, states_1, states_2, t_rin
g, g_use_fftw
    if status:
        print(status)
    # 입력 오디오 다운샘플링
    indata_resampled = resample_audio(indata.squeeze(), 441
00, 16000)
    # 이후 처리 로직...

    # 처리 완료된 오디오 업샘플링
    outdata_resampled = resample_audio(out_buffer[:block_sh
```

```
ift], 16000, 44100)
    outdata[:] = outdata_resampled.reshape(-1, 1)
```

- 실행결과 → 오류 발생

```
INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
Expression 'paInvalidSampleRate' failed in 'src/hostapi/alsa/
Expression 'PaAlsaStreamComponent_InitialConfigure( &self->ca
Expression 'PaAlsaStream_Configure( stream, inputParameters, 
PortAudioError: Error opening Stream: Invalid sample rate [Pa
```

코드 수정 과정

ns_0429.py

실시간 오디오 출력은 되지만, 노이즈 제거가 되지 않는다.

```
import numpy as np
import sounddevice as sd
import tflite_runtime.interpreter as tflite
import argparse
import collections
import time
import daemon
import threading
import scipy.signal

g_use_fftw = True

try:
    import pyfftw
except ImportError:
    print("[WARNING] pyfftw is not installed, use np.fft")
    g_use_fftw = False

def int_or_str(text):
    """Helper function for argument parsing."""
```

```
    try:
        return int(text)
    except ValueError:
        return text

parser = argparse.ArgumentParser(description=__doc__, formatt
parser.add_argument('-l', '--list-devices', action='store_tru
parser.add_argument('-i', '--input-device', type=int_or_str,
parser.add_argument('-o', '--output-device', type=int_or_str,
parser.add_argument('-c', '--channel', type=int, default=None
parser.add_argument('-n', '--no-denoise', action='store_true'
parser.add_argument('-t', '--threads', type=int, default=1, h
parser.add_argument('--latency', type=float, default=0.0001,
parser.add_argument('-D', '--daemonize', action='store_true',
parser.add_argument('--measure', action='store_true', help='m
parser.add_argument('--no-fftw', action='store_true', help='u

args = parser.parse_args()

fs_target = 16000  # Model sample rate
fs_device = 44100  # Device sample rate
block_len_ms = 32
block_shift_ms = 8

# 디바이스와 모델의 샘플레이트에 따라 적절히 조정
block_shift_device = int(np.round(fs_device * (block_shift_ms
block_len_device = int(block_shift_device * 4)  # 블록 길이는 시

block_shift = int(np.round(fs_target * (block_shift_ms / 1000
block_len = int(block_shift * 4)  # 블록 길이는 시프트의 4배로 설정

in_buffer_device = np.zeros((block_len_device)).astype('float
out_buffer_device = np.zeros((block_len_device)).astype('floa

in_buffer = np.zeros((block_len)).astype('float32')
out_buffer = np.zeros((block_len)).astype('float32')

interpreter_1 = tflite.Interpreter(model_path='./models/dtln_
```

```
interpreter_1.allocate_tensors()
interpreter_2 = tflite.Interpreter(model_path='./models/dtln_
interpreter_2.allocate_tensors()

input_details_1 = interpreter_1.get_input_details()
output_details_1 = interpreter_1.get_output_details()
input_details_2 = interpreter_2.get_input_details()
output_details_2 = interpreter_2.get_output_details()

states_1 = np.zeros(input_details_1[1]['shape']).astype('floa
states_2 = np.zeros(input_details_2[1]['shape']).astype('floa

def resample(data, orig_sr, target_sr):
    return scipy.signal.resample_poly(data, target_sr, orig_s

def callback(indata, outdata, frames, time, status):
    if status:
        print(status)
    global in_buffer, out_buffer, states_1, states_2
    if args.no_denoise:
        outdata[:] = indata
        return

    indata_resampled = resample(indata[:, 0], fs_device, fs_t
    in_buffer[:-block_shift] = in_buffer[block_shift:]
    in_buffer[-block_shift:] = indata_resampled[:block_shift]

    # FFT processing, DTLN model application, etc.
    # Simulated processing code here
    out_buffer[:-block_shift] = out_buffer[block_shift:]
    out_buffer[-block_shift:] = indata_resampled[:block_shift

    outdata_resampled = resample(out_buffer[:block_shift], fs_
    outdata[:, 0] = outdata_resampled[:frames]
    outdata[:, 1] = outdata_resampled[:frames]  # Assuming st

def open_stream():
    with sd.Stream(device=(args.input_device, args.output_dev
```

```
                    samplerate=fs_device, blocksize=block_shif
                    dtype='float32', channels=(1, 2), callback=
        print('#' * 80)
        print('Ctrl-C to exit')
        print('#' * 80)
        threading.Event().wait()

if __name__ == '__main__':
    if args.daemonize:
        with daemon.DaemonContext():
            open_stream()
    else:
        open_stream()
```

ns_0430.py

실시간 오디오 출력 및 외부 노이즈는 제거 되었지만, 음성에서 노이즈 발생

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Script to process realtime audio with a trained DTLN model.
This script supports ALSA audio devices. The model expects 16

Example call:
    $python rt_dtln_ns.py -i capture -o playback

Author: sanebow (sanebow@gmail.com)
Version: 23.05.2021

This code is licensed under the terms of the MIT-license.
"""



import numpy as np
import sounddevice as sd
import tflite_runtime.interpreter as tflite
import argparse
import collections
```

```
import time
import daemon
import threading

g_use_fftw = True

try:
    import pyfftw
except ImportError:
    print("[WARNING] pyfftw is not installed, use np.fft")
    g_use_fftw = False


def int_or_str(text):
    """Helper function for argument parsing."""
    try:
        return int(text)
    except ValueError:
        return text


parser = argparse.ArgumentParser(add_help=False)
parser.add_argument(
    '-l', '--list-devices', action='store_true',
    help='show list of audio devices and exit')
args, remaining = parser.parse_known_args()
if args.list_devices:
    print(sd.query_devices())
    parser.exit(0)
parser = argparse.ArgumentParser(
    description=__doc__,
    formatter_class=argparse.RawDescriptionHelpFormatter,
    parents=[parser])
parser.add_argument(
    '-i', '--input-device', type=int_or_str,
    help='input device (numeric ID or substring)')
parser.add_argument(
    '-o', '--output-device', type=int_or_str,
```

```
    help='output device (numeric ID or substring)')
parser.add_argument(
    '-c', '--channel', type=int, default=None,
    help='use specific channel of input device')
parser.add_argument(
    '-n', '--no-denoise', action='store_true',
    help='turn off denoise, pass-through')
parser.add_argument(
    '-t', '--threads', type=int, default=1,
    help='number of threads for tflite interpreters')
parser.add_argument(
    '--latency', type=float, default=0.0001,
    help='suggested input/output latency in seconds')
parser.add_argument(
    '-D', '--daemonize', action='store_true',
    help='run as a daemon')
parser.add_argument(
    '--measure', action='store_true',
    help='measure and report processing time')
parser.add_argument(
    '--no-fftw', action='store_true',
    help='use np.fft instead of fftw')


args = parser.parse_args(remaining)

# set some parameters
block_len_ms = 32
block_shift_ms = 8
fs_target = 16000

# create the interpreters
interpreter_1 = tflite.Interpreter(model_path='./models/dtln_
interpreter_1.allocate_tensors()
interpreter_2 = tflite.Interpreter(model_path='./models/dtln_
interpreter_2.allocate_tensors()
# Get input and output tensors.
input_details_1 = interpreter_1.get_input_details()
```

```
output_details_1 = interpreter_1.get_output_details()
input_details_2 = interpreter_2.get_input_details()
output_details_2 = interpreter_2.get_output_details()
# create states for the lstms
states_1 = np.zeros(input_details_1[1]['shape']).astype('floa
states_2 = np.zeros(input_details_2[1]['shape']).astype('floa
# calculate shift and length
block_shift = int(np.round(fs_target * (block_shift_ms / 1000
block_len = int(np.round(fs_target * (block_len_ms / 1000)))
# create buffer
in_buffer = np.zeros((block_len)).astype('float32')
out_buffer = np.zeros((block_len)).astype('float32')

if args.no_fftw:
    g_use_fftw = False
if g_use_fftw:
    fft_buf = pyfftw.empty_aligned(512, dtype='float32')
    rfft = pyfftw.builders.rfft(fft_buf)
    ifft_buf = pyfftw.empty_aligned(257, dtype='complex64')
    irfft = pyfftw.builders.irfft(ifft_buf)

t_ring = collections.deque(maxlen=100)



import librosa

# 샘플링 레이트 변환 함수
def resample_audio(data, original_sr, target_sr):
    return librosa.resample(data, orig_sr=original_sr, target_

def callback(indata, outdata, frames, buf_time, status):
    global in_buffer, out_buffer, states_1, states_2, t_ring,
    if args.measure:
        start_time = time.time()
    if status:
        print(status)

    # 입력 오디오 다운샘플링 (44100 -> 16000)
```

```
    indata_resampled = resample_audio(indata.squeeze(), 44100

    if args.channel is not None:
        indata_resampled = indata_resampled[:, [args.channel]

    if args.no_denoise:
        outdata[:] = indata
        if args.measure:
            t_ring.append(time.time() - start_time)
        return


    # write to buffer
    in_buffer[:-block_shift] = in_buffer[block_shift:]
    in_buffer[-block_shift:] = np.squeeze(indata)
    # calculate fft of input block
    if g_use_fftw:
        fft_buf[:] = in_buffer
        in_block_fft = rfft()
    else:
        in_block_fft = np.fft.rfft(in_buffer)
    in_mag = np.abs(in_block_fft)
    in_phase = np.angle(in_block_fft)
    # reshape magnitude to input dimensions
    in_mag = np.reshape(in_mag, (1,1,-1)).astype('float32')
    # set tensors to the first model
    interpreter_1.set_tensor(input_details_1[1]['index'], sta
    interpreter_1.set_tensor(input_details_1[0]['index'], in_
    # run calculation
    interpreter_1.invoke()
    # get the output of the first block
    out_mask = interpreter_1.get_tensor(output_details_1[0]['
    states_1 = interpreter_1.get_tensor(output_details_1[1]['
    # calculate the ifft
    estimated_complex = in_mag * out_mask * np.exp(1j * in_ph
    if g_use_fftw:
        ifft_buf[:] = estimated_complex
        estimated_block = irfft()
```

```
    else:
        estimated_block = np.fft.irfft(estimated_complex)
    # reshape the time domain block
    estimated_block = np.reshape(estimated_block, (1,1,-1)).a
    # set tensors to the second block
    interpreter_2.set_tensor(input_details_2[1]['index'], sta
    interpreter_2.set_tensor(input_details_2[0]['index'], est
    # run calculation
    interpreter_2.invoke()
    # get output tensors
    out_block = interpreter_2.get_tensor(output_details_2[0][
    states_2 = interpreter_2.get_tensor(output_details_2[1]['
    # write to buffer
    out_buffer[:-block_shift] = out_buffer[block_shift:]
    out_buffer[-block_shift:] = np.zeros((block_shift))
    out_buffer  += np.squeeze(out_block)
    # output to soundcard
    outdata[:] = np.expand_dims(out_buffer[:block_shift], axi
    if args.measure:
        t_ring.append(time.time() - start_time)

    # 처리 완료된 오디오 업샘플링 (16000 -> 44100)
    outdata_resampled = resample_audio(out_buffer[:block_shif
    # 업샘플된 데이터를 outdata 크기에 맞추어 잘라내기
    outdata_resampled = outdata_resampled[:outdata.shape[0]]
    outdata[:] = outdata_resampled.reshape(-1, 1)

def open_stream():
    with sd.Stream(device=(args.input_device, args.output_dev
                   samplerate=44100,  # 입력과 출력 샘플 레이트를
                   blocksize=block_shift,
                   dtype=np.float32, latency=args.latency,
                   channels=(1 if args.channel is None else N
        print('#' * 80)
        print('Ctrl-C to exit')
        print('#' * 80)
        if args.measure:
            while True:
```

```
                time.sleep(1)
                print('Processing time: {:.2f} ms'.format( 10
        else:
            threading.Event().wait()

try:
    if args.daemonize:
        with daemon.DaemonContext():
            open_stream()
    else:
        open_stream()
except KeyboardInterrupt:
    parser.exit('')
except Exception as e:
    parser.exit(type(e).__name__ + ': ' + str(e))
```

1. **라이브러리의 추가 및 변경**
    - ns_0430.py에서 `scipy.signal` 대신 `librosa` 라이브러리를 사용하여 오디오 데이터의 리샘플링을 처리하도록함.
2. **인자 파서의 변경**
    - ns_0430.py는 명령행 인자 파서에 `add_help=False` 옵션을 추가하여 초기 파서에서 도움말을 비활성화하고, 기존 파서를 부모로 추가 함. 이렇게 함으로써 도움말이 중복되는 것을 방지.
3. **오디오 처리 로직의 수정**
    - `callback` 함수 내에 FFT를 사용한 주파수 도메인 변환 및 처리 로직이 추가하고, 두 개의 모델을 차례로 사용하여 오디오 데이터를 처리하도록 변경 함.
4. **멀티스레딩 및 데몬화**
    - ns_0430.py에서는 `threading.Event().wait()`를 사용하여 스레드가 종료되지 않도록 관리함.

ns_0501.py

위의 코드를 합쳐, 실시간 오디오 출력 및 노이즈 제거를 진행하고자 하였으나 노이즈 제거가 제대로 이루어지지 않음

```
import numpy as np
import sounddevice as sd
import tflite_runtime.interpreter as tflite
import argparse
import collections
import time
import threading
import scipy.signal
import librosa

# FFTW 사용 여부
g_use_fftw = True

try:
    import pyfftw
except ImportError:
    g_use_fftw = False
    print("[WARNING] pyfftw is not installed, using np.fft")

def int_or_str(text):
    """Helper function for argument parsing."""
    try:
        return int(text)
    except ValueError:
        return text

parser = argparse.ArgumentParser(description=__doc__, formatt
parser.add_argument('-l', '--list-devices', action='store_tru
parser.add_argument('-i', '--input-device', type=int_or_str,
parser.add_argument('-o', '--output-device', type=int_or_str,
parser.add_argument('-c', '--channel', type=int, default=None
parser.add_argument('-n', '--no-denoise', action='store_true'
parser.add_argument('-t', '--threads', type=int, default=1, h
parser.add_argument('--latency', type=float, default=0.0001,
parser.add_argument('-D', '--daemonize', action='store_true',
parser.add_argument('--measure', action='store_true', help='m
parser.add_argument('--no-fftw', action='store_true', help='u
args = parser.parse_args()
```

```
if args.no_fftw:
    g_use_fftw = False

fs_target = 16000  # Model sample rate
fs_device = 44100  # Device sample rate
block_len_ms = 128
block_shift_ms = 8

block_shift = int(np.round(fs_target * (block_shift_ms / 1000
# block_len = fs_target * block_len_ms // 1000  # This must b

block_len = 512  # This is correctly set to 512
fft_size = block_len  # Make sure this assignment is consiste

if g_use_fftw:
    fft_buf = pyfftw.empty_aligned(fft_size, dtype='float32')
    rfft = pyfftw.builders.rfft(fft_buf)
    ifft_buf = pyfftw.empty_aligned(fft_size//2 + 1, dtype='c
    irfft = pyfftw.builders.irfft(ifft_buf)


in_buffer = np.zeros((block_len)).astype('float32')
out_buffer = np.zeros((block_len)).astype('float32')

interpreter_1 = tflite.Interpreter(model_path='./models/dtln_
interpreter_1.allocate_tensors()
interpreter_2 = tflite.Interpreter(model_path='./models/dtln_
interpreter_2.allocate_tensors()

input_details_1 = interpreter_1.get_input_details()
output_details_1 = interpreter_1.get_output_details()
input_details_2 = interpreter_2.get_input_details()
output_details_2 = interpreter_2.get_output_details()

states_1 = np.zeros(input_details_1[1]['shape']).astype('floa
states_2 = np.zeros(input_details_2[1]['shape']).astype('floa
```

```
def resample_audio(data, original_sr, target_sr, target_sampl
    resampled_data = scipy.signal.resample(data, target_sampl
    return resampled_data

#args.latency = 0.3  # 레이턴시를 더 늘려보세요. 0.2초에서 0.3초로 조

def callback(indata, outdata, frames, buf_time, status):
    if status:
        print("Status:", status)
    if args.no_denoise:
        outdata[:] = indata
        return

    # Process indata
    indata_resampled = resample_audio(indata[:, 0], fs_device
    in_buffer[:-block_shift] = in_buffer[block_shift:]
    in_buffer[-block_shift:] = indata_resampled[:block_shift]

    # FFT and Processing
    if g_use_fftw:
        fft_buf[:] = in_buffer
        in_block_fft = rfft()
    else:
        in_block_fft = np.fft.rfft(in_buffer, n=block_len)

    in_mag = np.abs(in_block_fft)
    in_mag = np.reshape(in_mag, (1, 1, -1)).astype('float32')

    interpreter_1.set_tensor(input_details_1[0]['index'], in_
    interpreter_1.invoke()
    out_mask = interpreter_1.get_tensor(output_details_1[0]['
    out_mask = np.reshape(out_mask, (1, 1, -1))

    phase = np.angle(in_block_fft)
    estimated_complex = in_mag * out_mask * np.exp(1j * phase

    if g_use_fftw:
        ifft_buf[:] = estimated_complex
```

```
        estimated_block = irfft()
    else:
        estimated_block = np.fft.irfft(estimated_complex, n=5

    out_block = np.squeeze(estimated_block)
    if out_block.shape[0] != block_shift:
        out_block = np.resize(out_block, (block_shift,))

    out_buffer[:-block_shift] = out_buffer[block_shift:]
    out_buffer[-block_shift:] = out_block

    outdata_resampled = resample_audio(out_buffer[:block_shif
    outdata[:, :] = outdata_resampled.reshape(-1, outdata.sha

# 오디오 출력을 담당하는 함수
def play_audio(output_data):
    sd.play(output_data, samplerate=fs_device, device=args.ou

# 실행 시 사용하는 오디오 디바이스 ID 확인을 위한 코드 추가
if args.list_devices:
    print(sd.query_devices())
    exit()

# 메인 실행 루프
# 메인 실행 블록 안에서 변수 초기화
if __name__ == '__main__':
    input_channels = args.channel if args.channel is not None
    output_channels = 2  # 일반적으로 스테레오 출력 설정

    try:
        with sd.Stream(device=(args.input_device, args.output_
                       samplerate=fs_device,
                       blocksize=block_len,
                       dtype='float32',
                       channels=(input_channels, output_chann
                       latency=args.latency,
                       callback=callback):
            print('#' * 80)
```

```
            print('Press Ctrl-C to exit')
            print('#' * 80)
            threading.Event().wait()  # 사용자가 스트림을 중단할 때
    except Exception as e:
        print(f"Error occurred: {e}")
```

1. **블록 길이의 변경**
    - ns_0501.py에서는 블록 길이(`block_len`)를 고정 값 512로 설정하며, FFT 사이즈도 이에 맞춰 고정 → FFT 처리의 효율성을 높이기 위한 것으로, 고정된 크기를 사용하는 것이 일반적으로 더 최적화된 성능 제공
2. **리샘플링 함수의 인자 추가**
    - ns_0501.py의 `resample_audio` 함수는 대상 샘플의 수(`target_samples`)를 인자로 받음 → 리샘플링 과정에서 더 정밀한 제어 가능
3. **추가된 오디오 재생 기능**
    - ns_0501.py에는 오디오 데이터를 재생하는 `play_audio` 함수를 추가하여, `open_stream()` 대신 처리된 오디오 데이터를 실시간으로 듣는 기능 제공

🤷‍♀️ PiDTLN을 활용하기 위해서는,,,

- PiDTLN의 ns.py 코드는 16000Hz를 대상으로 구현이 되어 있기에, 우리가 사용하는 오디오 장치의 주파수에 맞춰서 수정을 해야한다.
- 44100Hz의 입력을 16000Hz로 다운샘플링 후, 다시 44100Hz로 업샘플링하여 출력해야 한다.

⇒ 이 과정에서 잡음이 지속적으로 발생한다.

TODO

💡 PiDTLN 코드 수정을 보류하고, 이전에 작성한 실시간 노이즈 제거 코드를 활용하여 개발을 다시 진행하고자 한다.

- **실시간 노이즈 제거**

- [ ] 노이즈 제거 프레임 수 조정 → 수동 종료 전까지 노이즈 제거 진행 되도록 수정
- [ ] 딜레이 해결 → 실시간 오디오 출력 및 노이즈 제거 되도록 수정

- **RMS 계산** (실시간 노이즈 제거 코드 활용)
  - [ ] 두 개의 오디오 실시간 입출력 테스트(이후 4개까지 진행)
  - [ ] 단일 음성 RMS 실시간 연산
  - [ ] 다중 음성 평균 RMS 실시간 연산
  - [ ] 출력 latency 조정

실시간 노이즈 제거 및 실시간 오디오 출력

- RT_Noise.py
  - [ ] 딜레이 발생 ⇒ CHUNK 수 변경 (256 → 2)
  - [ ] 노이즈 프로파일링 기간 설정을 통해서만 특정 시간 내에서 노이즈 제거

    ⇒ 프로그램이 종료되기 전까지 연속적으로 노이즈 제거 처리가 되도록 해야 한다.
  - [ ] 이어폰 하나로 단일 동일 입출력 시 노이즈 제거

  ⇒ 사운드 카드를 통해 단일 입출력을 마이크와 이어폰 스피커로 따로 설정할 시 노이즈 발생

    ⇒ 외부 말소리 및 잡음 발생

```
import pyaudio
import numpy as np
import time

# 포맷, 채널 수, 샘플링 레이트, 청크 크기 설정
FORMAT = pyaudio.paFloat32
CHANNELS = 1
RATE = 44100
CHUNK = 256  # 한 프레임당 샘플 수

# RMS(Root Mean Square)를 계산하는 함수
def calculate_rms(audio):
```

```
    return np.sqrt(np.mean(np.square(audio)))

# 음량을 조절하는 함수
def adjust_volume(audio, target_rms, current_rms):
    if current_rms == 0:
        return audio
    adjustment_factor = target_rms / current_rms
    return audio * adjustment_factor

# 스펙트럴 서브트랙션을 통한 노이즈 제거 함수
def spectral_subtraction(signal, noise_estimation, alpha=4
    transformed_signal = np.fft.rfft(signal)
    subtracted_spectrum = np.maximum(transformed_signal - 
    cleaned_signal = np.fft.irfft(subtracted_spectrum, n=l
    return cleaned_signal

# 오디오 처리를 위한 클래스
class AudioProcessor:
    def __init__(self, target_rms=0.02, update_interval=10
        self.noise_profile = None
        self.initial_noise_data = []
        self.initial_noise_frames = int(RATE / CHUNK * 999
        self.update_frames = int(RATE / CHUNK * update_int
        self.frames_processed = 0
        self.target_rms = target_rms

    # 노이즈 프로파일 추정 함수
    def estimate_noise_profile(self, audio_data, force_upd
        self.initial_noise_data.append(audio_data)
        self.frames_processed += 1

        if len(self.initial_noise_data) >= self.initial_no
            noise_data = np.concatenate(self.initial_noise_
            self.noise_profile = np.abs(np.fft.rfft(noise_
            self.initial_noise_data = []  # 데이터 초기화

        # 주기적으로 노이즈 프로파일 업데이트
        if self.frames_processed % self.update_frames == 0
```

```
            self.estimate_noise_profile(audio_data, force_

    # 오디오 데이터 처리 함수
    def process_audio_data(self, in_data):
        audio_data = np.frombuffer(in_data, dtype=np.float
        if self.noise_profile is None or self.frames_proce
            self.estimate_noise_profile(audio_data)
            return in_data  # 충분한 노이즈 프로파일이 설정되기 전
        else:
            cleaned_data = spectral_subtraction(audio_data
            current_rms = calculate_rms(cleaned_data)
            adjusted_data = adjust_volume(cleaned_data, se
            return adjusted_data.astype(np.float32).tobyte

# 스트림 콜백 함수
def stream_callback(in_data, frame_count, time_info, statu
    adjusted_data = processor.process_audio_data(in_data)
    return (adjusted_data, pyaudio.paContinue)

processor = AudioProcessor(target_rms=0.02)  # RMS 목표값 설

def main():
    pa = pyaudio.PyAudio()
    stream = pa.open(format=FORMAT, channels=CHANNELS, rat
    stream.start_stream()

    try:
        while stream.is_active():
            time.sleep(0.1)
    except KeyboardInterrupt:
        print("사용자에 의해 스트림이 중단되었습니다.")
    finally:
        stream.stop_stream()
        stream.close()
        pa.terminate()

if __name__ == "__main__":
    main()
```

⇒ window 환경에서는 실행이 되지만, Linux 환경에서는 ALSA 관련 오류로 실행이 되지 않는다.

⇒ Linux 환경에서 실행되도록 코드 재작성

RealTime_Noise.py on RasberryPi

```
#RT_Noise.py

import pyaudio
import numpy as np
import time

FORMAT = pyaudio.paFloat32
CHANNELS = 1
RATE = 44100
CHUNK = 256  # 프레임 당 샘플 수

def calculate_rms(audio):
    return np.sqrt(np.mean(np.square(audio)))

def adjust_volume(audio, target_rms, current_rms):
    if current_rms == 0:
        return audio
    adjustment_factor = target_rms / current_rms
    return audio * adjustment_factor

def spectral_subtraction(signal, noise_estimation, alpha=4):
    transformed_signal = np.fft.rfft(signal)
    subtracted_spectrum = np.maximum(transformed_signal - alp
    cleaned_signal = np.fft.irfft(subtracted_spectrum, n=len(
    return cleaned_signal

class AudioProcessor:
    def __init__(self, target_rms=0.02, update_interval=10):
        self.noise_profile = None
        self.initial_noise_data = []
        self.initial_noise_frames = int(RATE / CHUNK * 999999
        self.update_frames = int(RATE / CHUNK * update_interv
```

```
        self.frames_processed = 0
        self.target_rms = target_rms

    def estimate_noise_profile(self, audio_data, force_update
        self.initial_noise_data.append(audio_data)
        self.frames_processed += 1

        if len(self.initial_noise_data) >= self.initial_noise
            noise_data = np.concatenate(self.initial_noise_da
            self.noise_profile = np.abs(np.fft.rfft(noise_dat
            self.initial_noise_data = []  # 데이터 클리어

        # 주기적으로 노이즈 프로파일 업데이트
        if self.frames_processed % self.update_frames == 0:
            self.estimate_noise_profile(audio_data, force_upd

    def process_audio_data(self, in_data):
        audio_data = np.frombuffer(in_data, dtype=np.float32)
        if self.noise_profile is None or self.frames_processe
            self.estimate_noise_profile(audio_data)
            return in_data  # 노이즈 프로파일이 충분히 설정되기 전에는
        else:
            cleaned_data = spectral_subtraction(audio_data, s
            current_rms = calculate_rms(cleaned_data)
            adjusted_data = adjust_volume(cleaned_data, self.
            return adjusted_data.astype(np.float32).tobytes()


def stream_callback(in_data, frame_count, time_info, status):
    adjusted_data = processor.process_audio_data(in_data)
    return (adjusted_data, pyaudio.paContinue)

processor = AudioProcessor(target_rms=0.02)  # RMS 값을 여기에서

def main():
    pa = pyaudio.PyAudio()
    stream = pa.open(format=FORMAT, channels=CHANNELS, rate=RA
    stream.start_stream()
```

```
    try:
        while stream.is_active():
            time.sleep(0.1)
    except KeyboardInterrupt:
        print("Stream stopped by user.")
    finally:
        stream.stop_stream()
        stream.close()
        pa.terminate()

if __name__ == "__main__":
    main()
```

RT_Noise.py를 라즈베리파이에서 실행시, 정상 실행되지만 AISA 서버 관련 오류발생.

⇒ AISA경고는 실행과는 무관한 오류

두 개의 실시간 입출력

- realtime_audio_stream.py

```
import pyaudio
import numpy as np
import time

# PyAudio 설정 상수
FORMAT = pyaudio.paFloat32  # 오디오 데이터 포맷 (32비트 float)
CHANNELS = 1                # 채널 수 (모노)
RATE = 44100                # 샘플 레이트 (Hz)
CHUNK = 1024                # 버퍼의 프레임 수 (각 콜백에 대해 처

class AudioProcessor:
    def __init__(self):
        """오디오 처리기 초기화"""
        pass

    def process_audio_data(self, in_data):
        """오디오 데이터 처리 메서드
        Args:
```

```
            in_data (bytes): 처리할 원시 오디오 데이터
        Returns:
            bytes: 처리된 오디오 데이터
        """
        return in_data  # 현재 구현에서는 입력 데이터를 그대로 반환

def stream_callback(processor, in_data, frame_count, time_info, status):
    """스트림 콜백 함수
    Args:
        processor (AudioProcessor): 오디오 처리를 담당할 프로세서 객체
        in_data (bytes): 원시 오디오 데이터
        frame_count (int): 이 콜백에서 처리할 프레임 수
        time_info (dict): 타임스탬프 및 기타 타임 정보 포함
        status (int): 스트림 상태 플래그
    Returns:
        tuple: (처리된 오디오 데이터, 플래그)
    """
    adjusted_data = processor.process_audio_data(in_data)
    return adjusted_data, pyaudio.paContinue  # 처리된 데이터와 계속 진행 플래그 반환

processor1 = AudioProcessor()  # 첫 번째 오디오 프로세서 인스턴스 생성
processor2 = AudioProcessor()  # 두 번째 오디오 프로세서 인스턴스 생성

def main():
    pa = pyaudio.PyAudio()  # PyAudio 인스턴스 생성
    # 첫 번째 오디오 스트림 설정 및 생성
    stream1 = pa.open(format=FORMAT, channels=CHANNELS, rate=RATE, input=True, output=True,
                      frames_per_buffer=CHUNK, stream_callback=lambda in_data, frame_count, time_info, status: stream_callback(processor1, in_data, frame_count, time_info, status),
                      input_device_index=8, output_device_index=4)
    # 두 번째 오디오 스트림 설정 및 생성
    stream2 = pa.open(format=FORMAT, channels=CHANNELS, rate=RATE, input=True, output=True,
                      frames_per_buffer=CHUNK, stream_callback=lambda in_data, frame_count, time_info, status: stream_callback(processor2, in_data, frame_count, time_info, status),
                      input_device_index=9, output_device_index=5)

    stream1.start_stream()  # 첫 번째 스트림 시작
    stream2.start_stream()  # 두 번째 스트림 시작

    try:
```

```
        # 스트림이 활성 상태인 동안 대기
        while stream1.is_active() and stream2.is_active():
            time.sleep(0.1)
    except KeyboardInterrupt:
        # 사용자에 의한 중단 처리
        print("Stream stopped by user.")
    finally:
        # 스트림 및 리소스 정리
        stream1.stop_stream()
        stream1.close()
        stream2.stop_stream()
        stream2.close()
        pa.terminate()  # PyAudio 세션 종료

if __name__ == "__main__":
    main()
```

두 개의 독립적인 입력 소스에서 오디오 데이터를 받아 실시간으로 처리하고, 동일한 디바이스에 출력하도록 구현.

10초간 두 개의 실시간 입출력을 통해 각 RMS 및 평균 RMS 구하기

- realtime_rms_calculate.py

```
import pyaudio
import numpy as np
import time

# PyAudio 설정 상수
FORMAT = pyaudio.paFloat32  # 오디오 데이터 포맷 (32비트 float)
CHANNELS = 1                # 채널 수 (모노)
RATE = 44100                # 샘플 레이트 (Hz)
CHUNK = 1024                # 프레임 당 샘플 수

DURATION = 10  # 측정할 시간(초)

class AudioProcessor:
    """ 오디오 데이터를 처리하여 RMS 값을 계산하고 저장하는 클래스 """
    def __init__(self):
```

```
        self.rms_values = []  # RMS 값을 저장할 리스트

    def process_audio_data(self, in_data):
        """ 오디오 데이터로부터 RMS 값을 계산하고 저장하는 함수
        Args:
            in_data (bytes): 입력 오디오 데이터 (원시 바이트 데이
        Returns:
            bytes: 처리된 오디오 데이터 (이 경우 입력 데이터 그대로
        """
        # 바이트 데이터를 numpy 배열로 변환
        audio_data = np.frombuffer(in_data, dtype=np.float
        # RMS 계산: 데이터 제곱의 평균에 루트를 취함
        rms = np.sqrt(np.mean(np.square(audio_data)))
        # 계산된 RMS 값을 리스트에 추가
        self.rms_values.append(rms)
        return in_data  # 스트림을 위해 입력 데이터 반환

    def get_average_rms(self):
        """ 저장된 RMS 값들의 평균을 계산하여 반환 """
        return np.mean(self.rms_values)

def stream_callback(processor, in_data, frame_count, time_
    """ 오디오 스트림의 콜백 함수
    Args:
        processor (AudioProcessor): 오디오 처리기 인스턴스
        in_data (bytes): 입력 오디오 데이터
        frame_count (int): 이 콜백에서 처리할 프레임 수
        time_info (dict): 스트림 관련 시간 정보
        status (int): 스트림 상태 정보
    Returns:
        tuple: (처리된 오디오 데이터, pyaudio.paContinue)
    """
    # 오디오 데이터 처리
    adjusted_data = processor.process_audio_data(in_data)
    # 처리된 데이터와 처리 계속 진행 플래그 반환
    return adjusted_data, pyaudio.paContinue

def main():
```

```
    pa = pyaudio.PyAudio()  # PyAudio 인스턴스 생성

    # 오디오 프로세서 인스턴스 생성
    processor1 = AudioProcessor()
    processor2 = AudioProcessor()

    # 오디오 스트림 설정 및 생성
    stream1 = pa.open(format=FORMAT, channels=CHANNELS, ra
                      frames_per_buffer=CHUNK, stream_call
                      input_device_index=8, output_device_
    stream2 = pa.open(format=FORMAT, channels=CHANNELS, ra
                      frames_per_buffer=CHUNK, stream_call
                      input_device_index=9, output_device_

    # 스트림 시작
    stream1.start_stream()
    stream2.start_stream()

    try:
        # DURATION 동안 스트림 활성화 상태 유지
        start_time = time.time()
        while time.time() - start_time < DURATION:
            time.sleep(0.1)  # 작은 대기 시간으로 CPU 사용을 제

        # 스트림 종료 후 RMS 평균 값 계산 및 출력
        average_rms1 = processor1.get_average_rms()
        average_rms2 = processor2.get_average_rms()
        average_rms = (average_rms1 + average_rms2) / 2
        print("Average RMS of Stream 1:", average_rms1)
        print("Average RMS of Stream 2:", average_rms2)
        print("Average RMS of both streams:", average_rms)

    except KeyboardInterrupt:
        # 사용자에 의한 중단 처리
        print("Stream stopped by user.")
    finally:
        # 스트림 및 리소스 정리
        stream1.stop_stream()
```

```
        stream1.close()
        stream2.stop_stream()
        stream2.close()
        pa.terminate()  # PyAudio 세션 종료

if __name__ == "__main__":
    main()
```

realtime_audio_stream.py 코드에서 실시간으로 각 스트림의 rms를 계산하고, 두 스트림의 평균 rms를 계산하여 출력.

10초간 두 개의 실시간 입출력을 통해 각 RMS 및 평균 RMS 구한 후, 실시간 출력을 평균 RMS에 맞춰서 출력하기

- audio_rms_normalization.py

```
import pyaudio
import numpy as np
import time

# 오디오 스트림 설정 상수
FORMAT = pyaudio.paFloat32  # 32비트 부동 소수점 포맷
CHANNELS = 1                # 싱글 채널 (모노)
RATE = 44100                # 샘플링 레이트 (44.1kHz)
CHUNK = 1024                # 프레임 당 샘플 수 (버퍼 크기)
DURATION = 10               # 초기 RMS 계산을 위한 측정 시간 (초

class AudioProcessor:
    """ 오디오 신호의 RMS를 계산하고, 필요에 따라 신호를 조정하는 클래
    def __init__(self):
        self.rms_values = []  # 계산된 RMS 값들을 저장하는 리스트
        self.target_rms = None  # 조정 대상 RMS 값

    def process_audio_data(self, in_data, adjust_rms=False
        """ 오디오 데이터 처리 및 RMS 계산. 필요시 RMS를 조정하여 반
        audio_data = np.frombuffer(in_data, dtype=np.float
        rms = np.sqrt(np.mean(np.square(audio_data)))  # 현
        self.rms_values.append(rms)  # RMS 값 저장
```

```
        if adjust_rms and self.target_rms is not None:
            # RMS를 기반으로 scale 조정
            scale = (self.target_rms / (rms + 1e-10))**0.5
            adjusted_data = (audio_data * scale).astype(np
            return adjusted_data
        return in_data

    def set_target_rms(self, target_rms):
        """ 목표 RMS 값을 설정 """
        self.target_rms = target_rms

    def get_average_rms(self):
        """ 저장된 RMS 값들의 평균을 계산하여 반환 """
        return np.mean(self.rms_values)

def stream_callback(processor, adjust_rms, in_data, frame_
    """ PyAudio 스트림 콜백 함수. 오디오 데이터를 처리하고, 스트림을
    adjusted_data = processor.process_audio_data(in_data, 
    return adjusted_data, pyaudio.paContinue

def main():
    pa = pyaudio.PyAudio()  # PyAudio 인스턴스 생성
    processor1 = AudioProcessor()  # 첫 번째 프로세서
    processor2 = AudioProcessor()  # 두 번째 프로세서

    # 두 개의 스트림 설정 및 생성
    stream1 = pa.open(format=FORMAT, channels=CHANNELS, ra
                      frames_per_buffer=CHUNK, stream_call
                      input_device_index=8, output_device_
    stream2 = pa.open(format=FORMAT, channels=CHANNELS, ra
                      frames_per_buffer=CHUNK, stream_call
                      input_device_index=9, output_device_

    stream1.start_stream()  # 첫 번째 스트림 시작
    stream2.start_stream()  # 두 번째 스트림 시작

    try:
```

```
        # RMS 계산 및 조정 과정
        start_time = time.time()
        while time.time() - start_time < DURATION:
            time.sleep(0.1)

        # 평균 RMS 값 계산
        average_rms1 = processor1.get_average_rms()
        average_rms2 = processor2.get_average_rms()
        average_rms = (average_rms1 + average_rms2) / 2
        print("Average RMS of Stream 1:", average_rms1)
        print("Average RMS of Stream 2:", average_rms2)
        print("Average RMS of both streams:", average_rms)

        # 스트림에 RMS 조정 적용
        processor1.set_target_rms(average_rms)
        processor2.set_target_rms(average_rms)
        stream1.stop_stream()
        stream1.close()
        stream2.stop_stream()
        stream2.close()

        # 조정된 RMS 값으로 새 스트림 시작
        stream1 = pa.open(format=FORMAT, channels=CHANNELS
                          frames_per_buffer=CHUNK, stream_
                          input_device_index=8, output_dev
        stream2 = pa.open(format=FORMAT, channels=CHANNELS
                          frames_per_buffer=CHUNK, stream_
                          input_device_index=9, output_dev
        stream1.start_stream()
        stream2.start_stream()

        print("RMS adjusted streams started. Press Ctrl+C
        while True:
            time.sleep(0.1)

    except KeyboardInterrupt:
        # 사용자 중단 처리
        print("Stream stopped by user.")
```

```
    finally:
        # 스트림 및 리소스 정리
        stream1.stop_stream()
        stream1.close()
        stream2.stop_stream()
        stream2.close()
        pa.terminate()

if __name__ == "__main__":
    main()
```

두 오디오 스트림의 실시간 데이터를 처리하여 초기 10초 동안 RMS 값을 계산.

계산된 평균 RMS를 바탕으로, 이후 오디오 데이터의 볼륨을 평균 RMS 수준으로 조정.

TODO

💡 두 스트림의 평균 rms값을 기반으로 오디오 데이터 볼륨 조절이 성공적으로 되었는지 확인하기 위해 시각자료 출력이 필요함.

- **그래프 출력 함수 구현**
  - ☐ 입력된 음성을 그래프로 어떻게 보여줄건지(x축, y축 등 기준값 선정)
  - ☐ input data를 wav파일로 저장하여 그래프로 변환 (2개의 audio 동시 입력)
  - ☐ normalized data를 wav파일로 저장하여 그래프로 변환 (2개의 audio 동시 입력)
  - ☐ 실시간으로 input data의 rms를 그래프로 출력 (2개의 audio 동시 입력)
- **노이즈제거 성능 검증 필요**

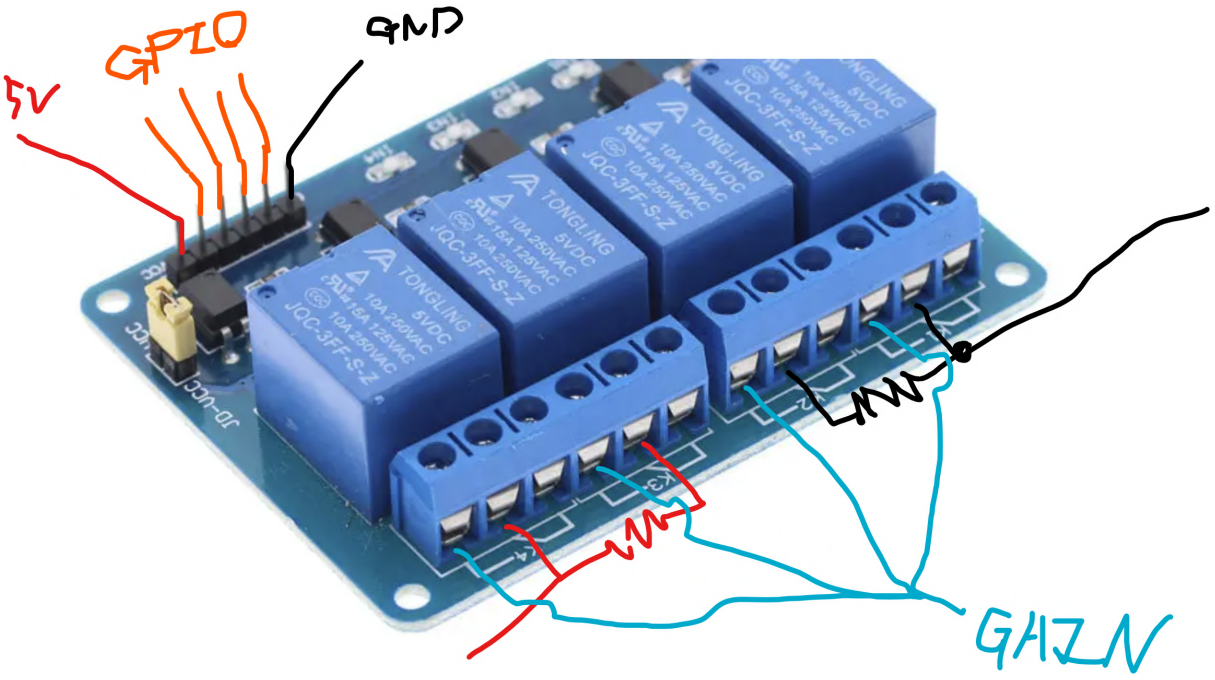
<하드웨어>

1️⃣ 4/29

MAX98375a 모듈은 5개의 증폭도를 가지고 있는데, 이는 회로를 통해서만 선택이 가능하다. 따라서 릴레이를 이용해서 라즈베리파이에서의 GPIO제어를 통해 특정회로를 선택할 수 있도록 설계함

Gain Selection

GAIN_SLOT	GAIN
Connect to GND through 100kΩ resistor	+15 dB
Connect to GND	+12 dB
Unconnected (Default)	+9 dB
Connect to VDD	+6 dB
Connect to VDD through 100kΩ resistor	+3 dB



이때 모든 GPIO가 off되어 있으면 gain에 아무것도 연결하지 않는 상태가 되어 디폴트 값이 9dB이 증폭도를 가지고 오디오 신호가 증폭되도록 한다.

GPIO 제어

hw 제어만을 확인해보기 위해 라즈베리파이에 마이크를 통해서 입력한 소리 신호 한 개를, gpio제어를 통해 릴레이에서 특정채널을 선택하여 특정 증폭도로 신호를 증폭하여 스피커를 통해 출력하는 테스트를 진행하려고 함. 아직 모듈이 안 온 관계로 이때 필요한 제어 코드 먼저 작성함

1. 릴레이 모듈 설정

먼저, 릴레이 모듈의 각 채널을 설정하고, 라즈베리파이의 GPIO 핀을 통해 릴레이를 제어할 수 있는 코드 작성

```
# GPIO 라이브러리 설치
pip install RPi.GPIO
```

```
import RPi.GPIO as GPIO
import time
```

```
# 릴레이 모듈의 GPIO 핀 설정
relay_pins = [21, 22, 23, 24]

# GPIO 모드 설정
GPIO.setmode(GPIO.BCM)

# 릴레이 핀 설정 (출력 모드)
for pin in relay_pins:
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, GPIO.LOW)  # 기본값을 LOW로 설정
```

2. 15dB 증폭을 위해 릴레이 제어

릴레이 모듈의 2번 채널을 사용하여 증폭도 15dB를 선택한다고 가정, 해당 채널을 활성화하고 나머지 채널을 비활성화하는 코드

```
def set_gain_15db():
    # 모든 릴레이를 비활성화
    for pin in relay_pins:
        GPIO.output(pin, GPIO.LOW)

    # 2번 채널 (릴레이 22번 핀) 활성화
    GPIO.output(22, GPIO.HIGH)
```

3. 마이크로 오디오 입력 수신 및 I2S를 통한 전송

마이크에서 오디오 입력을 수신하고, 이를 I2S 프로토콜을 사용하여 증폭모듈로 전송(이때 해당 I2S프로토콜이 정확하지는 않음 → 아직 정확한 문서를 찾지 못함)

```
# I2S 관련 라이브러리 설치
pip install sounddevice
```

```
import sounddevice as sd
import numpy as np

# 오디오 입력 설정
sample_rate = 44100  # 샘플링 속도
duration = 5  # 5초 동안 오디오를 수신
```

```
# 오디오 데이터 수신 함수
def record_audio(duration, sample_rate):
    audio_data = sd.rec(int(duration * sample_rate), samplera
    sd.wait()
    return audio_data

# 오디오 수신
audio = record_audio(duration, sample_rate)
```

```
# 릴레이 설
set_gain_15db()
audio = record_audio(duration, sample_rate)
```

추가 이슈

- 100k옴 저항이 부족하여 추가 구매
- 증폭모듈 MAX98357a 배송 아직 안 옴

- 원활한 마이크 입력을 위해 중간에 사운드카드를 연결해서 사용하기보다 usb전용 마이크를 사용할지 고려

다음 할 일

- 증폭모듈 연결하여 실제 GPIO제어 확인하기
    - 이때 마이크 입력신호가 제대로 전달이 안되는 경우 오디오 파일을 이용해서 테스트 진행하기

(원래는, 입력신호의 노이즈 제거 → RMS → RMS에 따른 GPIO를 통한 증폭도 선택 → 증폭 → 출력 이 순으로 가야 하는 게 맞지만, 노이즈 제거와 RMS계산 부분이 아직 구현이 안되어, 하드웨어 부분만 테스트 진행 예정)

2️⃣ 4/30

I2S 활성화

라즈베리파이 5에서 I2S를 활성화하려면 이전 모델(3,4)과 유사하게 `config.txt` 파일에서 I2S를 활성화해야함

`dtoverlay=i2s-mmap` 설정이 일반적으로 사용되며, 라즈베리파이를 재부팅하여 적용함

라즈베리파이 5 I2S 핀 연결

라즈베리파이 5에서 I2S를 사용하려면 다음과 같은 핀을 사용한다. 일반적으로 라즈베리파이의 I2S 핀은 다음과 같이 할당됨

- **BCLK (비트 클럭)**: I2S 비트 클럭으로, 데이터 비트의 타이밍을 제어
- **LRCK (좌/우 채널 클럭)**: I2S 워드 클럭 또는 좌/우 채널 클럭으로, 오디오 스트림에서 채널구분
- **DIN (데이터 입력)**: 외부 기기로부터 데이터 입력을 수신
- **DOUT (데이터 출력)**: 라즈베리파이에서 데이터 출력을 전송

일반적으로 다음과 같은 핀이 I2S 인터페이스로 사용

- **BCLK**: GPIO 18 (BCM 기준)
- **LRCK**: GPIO 19
- **DIN**: GPIO 20
- **DOUT**: GPIO 21

→ 이부분은 라즈베리파이5의 경우 다를 수 있어서 다시 찾아봐야 한다. (정확하지 않음)

실시간 data 전송

1. 릴레이 모듈 제어 코드

릴레이 2번 채널을 선택하여 증폭도 15dB을 선택하는 경우, 해당 채널을 활성화하는 테스트 진행

```
import RPi.GPIO as GPIO

# GPIO 모드 설정
GPIO.setmode(GPIO.BCM)

# 릴레이 모듈의 GPIO 핀 설정 (예: GPIO 21, 22, 23, 24)
```

```
relay_pins = [21, 22, 23, 24]
for pin in relay_pins:
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, GPIO.LOW)  # 초기 상태는 LOW

# 증폭도 15dB 설정 (릴레이 2번 채널 활성화)
def set_gain_15db():
    for pin in relay_pins:
        GPIO.output(pin, GPIO.LOW)  # 모든 릴레이 비활성화
    GPIO.output(22, GPIO.HIGH)  # 2번 채널 활성화
```

2. 마이크로부터 실시간 오디오 수신

실시간으로 마이크로부터 오디오 데이터를 수신하려면, 오디오 입력을 지속적으로 수신하는 스트림(stream) 기반 방식이 필요하다. `pyaudio`나 `sounddevice`와 같은 라이브러리를 사용하여 마이크 입력을 실시간으로 읽는다

```
# 파이썬 오디오 라이브러리 설치
pip install sounddevice
```

```
import sounddevice as sd
import numpy as np

# 오디오 입력 설정
sample_rate = 44100  # 샘플링 속도
channels = 1  # 모노
chunk = 1024  # 버퍼 크기

# 실시간 스트리밍을 위한 콜백 함수
def audio_callback(indata, frames, time, status):
    if status:
        print(status, file=sys.stderr)
    # 오디오 데이터 처리를 위한 코드 작성
    # 이 부분에서 데이터를 I2S로 전송하거나 다른 프로세싱 수행

# 실시간 스트리밍 시작
stream = sd.InputStream(samplerate=sample_rate,
                        channels=channels,
```

```
                        callback=audio_callback,
                        blocksize=chunk)

with stream:
    print("Streaming...")
    # 스트리밍 동안 루프 실행
    while True:
        # 원하는 작업 수행
        pass
```

3. I2S를 통한 오디오 전송 코드

```
# config.txt에서 I2S 활성화
sudo nano /boot/config.txt
```

```
# I2S 활성화
dtoverlay=i2s-mmap
```

전체 통합

위의 각 코드를 통합

릴레이 제어(증폭도 선), 마이크 입력 수신, I2S를 통한 오디오 전송

```
import RPi.GPIO as GPIO
import sounddevice as sd

# 릴레이 설정
relay_pins = [21, 22, 23, 24]
GPIO.setmode(GPIO.BCM)
for pin in relay_pins:
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, GPIO.LOW)

# 증폭도 15dB 설정
```

```
def set_gain_15db():
    for pin in relay_pins:
        GPIO.output(pin, GPIO.LOW)
    GPIO.output(22, GPIO.HIGH)

set_gain_15db()  # 증폭도 15dB 선택

# 오디오 입력 설정
sample_rate = 44100  # 샘플링 속도
channels = 1  # 모노
chunk = 1024  # 버퍼 크기

# 실시간 스트리밍을 위한 콜백 함수
def audio_callback(indata, frames, time, status):
    if status:
        print(status, file=sys.stderr)
    # 여기에서 I2S를 통해 데이터를 전송하거나 다른 프로세싱 수행

# 실시간 스트리밍 시작
stream = sd.InputStream(samplerate=sample_rate,
                        channels=channels,
                        callback=audio_callback,
                        blocksize=chunk)

with stream:
    print("Streaming...")  # 스트리밍 시작
    while True:
        # 실시간 오디오 전송 및 추가 작업 수행
        pass
```

3️⃣ 05/01

GPIO 제어

오류

```
pi@raspberrypi:~/myenv/test $ sudo python3 0501.py
Traceback (most recent call last):
  File "/home/pi/myenv/test/0501.py", line 13, in <module>
    GPIO.setup(pin, GPIO.OUT)
RuntimeError: Cannot determine SOC peripheral base address
```

→ gpiozero를 이용하여 오류 해결

```
import sounddevice as sd
import gpiozero
import sys  # 'sys' 모듈 임포트

# 릴레이 모듈의 GPIO 핀 설정
relay_pins = [6, 13, 19, 26]

# 릴레이 핀을 gpiozero.OutputDevice로 설정
relays = [gpiozero.OutputDevice(pin) for pin in relay_pins]

# 기본값을 LOW로 설정
for relay in relays:
    relay.off()

def set_gain_6db():
    # 모든 릴레이를 비활성화
    for relay in relays:
        relay.on()

    # 6번 릴레이 활성화
    relays[0].off()  # 첫 번째 핀이 6번이므로 첫 번째 릴레이를 활성화

set_gain_6db()  # 증폭도 6dB 선택

# 오디오 입력 설정
sample_rate = 22050  # 샘플링 속도
channels = 1  # 모노
chunk = 4096  # 버퍼 크기
```

```
# 실시간 스트리밍을 위한 콜백 함수
def audio_callback(indata, frames, time, status):
    if status:
        print(status, file=sys.stderr)
    # I2S로 전송하거나 간단한 작업만 수행


# 실시간 스트리밍 시작
stream = sd.InputStream(samplerate=sample_rate,
                        channels=channels,
                        callback=audio_callback,
                        blocksize=chunk,
                        latency='high')  # 높은 지연 시간으로 설정


with stream:
    print("Streaming...")  # 스트리밍 시작
    while True:
        # 실시간 오디오 전송 및 추가 작업 수행
        pass
```

다음 할 일

증폭모듈의 출력을 테스트 할 4옴 3W출력 스피커가 없어서 제대로 릴레이 제어가 되는지 확인이 불가함

이를 시각적으로 확인하기 위해 led를 연결하여 릴레이가 제대로 제어가 되는 지 확인

- 테스트용 스피커가 도착하기 전까지 led로 릴레이 제어 확인
- 마이크 두 개 이상 연결 시도

4️⃣ 05/02

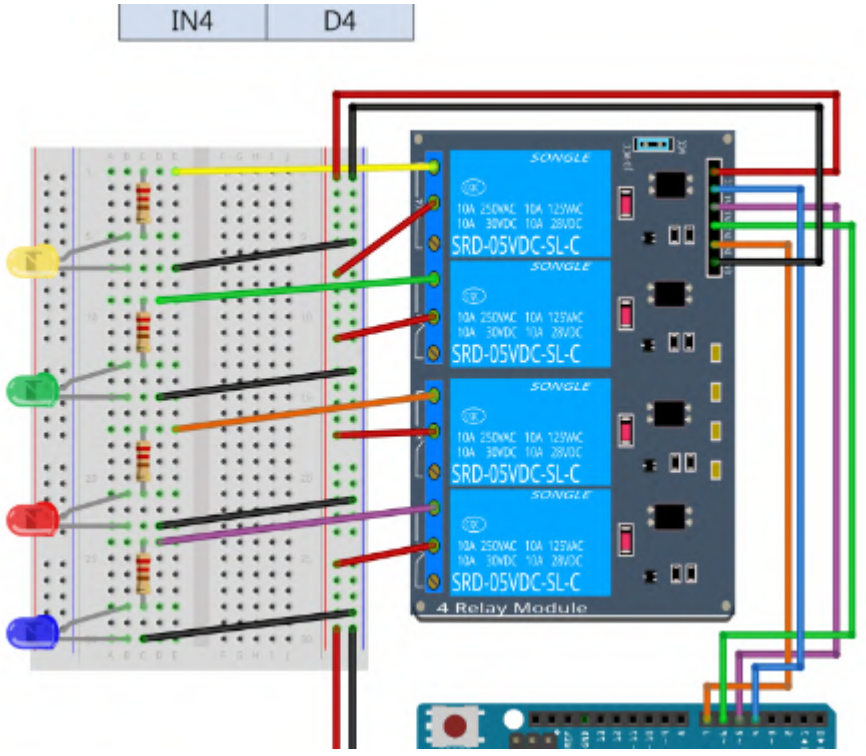
💡 그러면 우리가 지금 해야하는거는
마이크 2개의 입력을 파일로 만들자마자
해당 파일이 증폭 모듈로 가서 증폭해서 출력이 되는지 각각

근데 지금 스피커가 안 와서 테스트 불가

릴레이제어 - LED

기존에 증폭모듈 연결을 하여 연결이 되는 것까지는 확인하였으나 실제로 소리의 출력이 잘 되는지는 테스트용 스피커가 아직 없어서 확인 불가능

따라서 led를 채널마다 연결하여 릴레이 제어가 제대로 되는지 확인




```
from gpiozero import OutputDevice
from time import sleep

# 4채널 릴레이를 GPIO 6, 13, 19, 26에 연결
relay1 = OutputDevice(6)
relay2 = OutputDevice(13)
```

```
relay3 = OutputDevice(19)
relay4 = OutputDevice(26)

try:
    while True:
        # 릴레이를 순차적으로 켜고 끄기
        relay1.on()
        sleep(1)
        relay1.off()

        relay2.on()
        sleep(1)
        relay2.off()

        relay3.on()
        sleep(1)
        relay3.off()

        relay4.on()
        sleep(1)
        relay4.off()

except KeyboardInterrupt:
    # 프로그램 종료 시 모든 릴레이를off
    relay1.off()
    relay2.off()
    relay3.off()
    relay4.off()
    print("프로그램 종료")
finally:
    gpiozero.cleanup()  # 모든 GPIO 설정을 정리하고 해제
```

4채널 릴레이 두 개 제어

위에서는 릴레이 하나에서의 채널들 제어라면, 실제로 구현해야 하는 것은 2개 이상의 마이크가 동시 제어가 가능해야 하므로 2개 이상의 릴레이에서 각각 5개씩의 led를 연결하여 제어

라즙베리파이 - 증폭모듈연결

ttps://learn.adafruit.com/adafruit-max98357-i2s-class-d-mono-amp/raspberry-pi-usage

https://blog.naver.com/mapes_khkim/222532292451

마이크 동시 입력

- 쓰레드 마이크 동시 입력+녹음 → 파일 저장

```
import sounddevice as sd
import soundfile as sf
import threading
import time

# 마이크 및 스피커 장치 설정
mic1_device = 3  # 첫 번째 마이크 (Device 3: USB Audio Device)
mic2_device = 5  # 두 번째 마이크 (Device 5: USB Audio Device)

# 각 마이크의 채널 수
mic_channels = 1  # 마이크는 모노

# 스피커 장치 설정
speaker1_device = 2  # 첫 번째 스피커 (스테레오)
speaker2_device = 5  # 두 번째 스피커 (스테레오)

# 스피커의 채널 수
speaker_channels = 2  # 스테레오 출력

# 녹음 파일 설정
output_file1 = "mic1_recording.wav"
output_file2 = "mic2_recording.wav"

# 스트리밍 및 녹음 함수
```

```
def stream_and_record(input_device, output_device, input_chan
    # WAV 파일 생성 및 쓰기 모드로 열기
    with sf.SoundFile(output_file, mode='w', samplerate=44100
        def callback(indata, outdata, frames, time, status):
            if status:
                print(f"Status: {status}")
            # 입력 데이터를 WAV 파일로 녹음
            file.write(indata)
            # 입력 데이터를 출력으로 복사
            outdata[:] = indata
            # 로그 출력
            print(f"Streaming and recording {frames} frames f

        # 스트리밍 시작
        with sd.Stream(device=(input_device, output_device),
                       channels=(input_channels, output_chann
                       callback=callback):
            # 스트리밍이 지속되도록 한다
            while True:
                time.sleep(1)

# 쓰레드 생성 및 시작
thread1 = threading.Thread(target=stream_and_record,
                           args=(mic1_device, speaker1_device
thread2 = threading.Thread(target=stream_and_record,
                           args=(mic2_device, speaker2_device

# 쓰레드 시작
thread1.start()
thread2.start()

# 메인 스레드 대기
thread1.join()
thread2.join()
```

- 오디오 데이터를 실시간으로 터미널에 출력 + 녹음파일 생성

```
pip install sounddevice soundfile
```

```
import sounddevice as sd
import soundfile as sf
import threading
import time
import numpy as np

# 마이크 및 스피커 장치 설정
mic1_device = 3  # 첫 번째 마이크 (예시)
mic2_device = 5  # 두 번째 마이크 (예시)

# 각 마이크의 채널 수
mic_channels = 1  # 마이크는 모노

# 스피커 장치 설정
speaker1_device = 2  # 첫 번째 스피커 (스테레오)
speaker2_device = 5  # 두 번째 스피커 (스테레오)

# 스피커의 채널 수
speaker_channels = 2  # 스테레오 출력

# 녹음 파일 설정
output_file1 = "mic1_recording.wav"
output_file2 = "mic2_recording.wav"

# 스트리밍 및 녹음 함수
def stream_and_record(input_device, output_device, input_chan
    # WAV 파일 생성 및 쓰기 모드로 열기
    with sf.SoundFile(output_file, mode='w', samplerate=44100
        def callback(indata, outdata, frames, time, status):
            if status:
                print(f"Status: {status}")
            # 입력 데이터를 WAV 파일로 녹음
            file.write(indata)
            # 입력 데이터를 출력으로 복사
            outdata[:] = indata
```

```
            # 입력 데이터를 간략히 터미널에 출력
            print(f"Streaming {frames} frames from device {in
                  f"Min: {np.min(indata)}, Max: {np.max(indat
                  f"Mean: {np.mean(indata)}")

        # 스트리밍 시작
        with sd.Stream(device=(input_device, output_device),
                       channels=(input_channels, output_chann
                       callback=callback):
            # 스트리밍이 지속되도록 한다
            while True:
                time.sleep(1)

# 쓰레드 생성 및 시작
thread1 = threading.Thread(target=stream_and_record,
                           args=(mic1_device, speaker1_device
thread2 = threading.Thread(target=stream_and_record,
                           args=(mic2_device, speaker2_device

# 쓰레드 시작
thread1.start()
thread2.start()

# 메인 스레드 대기
thread1.join()
thread2.join()
```

- 쓰레드를 통한 마이크 동시 입력

```
import pyaudio
import numpy as np
import threading

# 파이오디오 객체 생성
p = pyaudio.PyAudio()

# 입력 설정
```

```
CHUNK = 512  # 더 작은 CHUNK 크기
FORMAT = pyaudio.paInt16
CHANNELS = 1
RATE = 44100

# 마이크와 스피커에 대한 장치 인덱스 설정
INPUT_DEVICE_INDICES = [0, 1]  # 마이크 장치 인덱스 리스트
OUTPUT_DEVICE_INDICES = [0, 1]  # 스피커 장치 인덱스 리스트

# 입력 스트림과 출력 스트림을 저장할 딕셔너리 생성
input_streams = {}
output_streams = {}

# 스트림 생성
for i in range(len(INPUT_DEVICE_INDICES)):
    input_streams[i] = p.open(
        format=FORMAT,
        channels=CHANNELS,
        rate=RATE,
        input=True,
        input_device_index=INPUT_DEVICE_INDICES[i],
        frames_per_buffer=CHUNK
    )

    output_streams[i] = p.open(
        format=FORMAT,
        channels=CHANNELS,
        rate=RATE,
        output=True,
        output_device_index=OUTPUT_DEVICE_INDICES[i],
        frames_per_buffer=CHUNK
    )

def stream_audio(input_stream, output_stream):
    """입력 스트림에서 오디오 데이터를 읽어 출력 스트림으로 보내는 함수"""
    try:
        while True:
            # 오디오 데이터 읽기
```

```
            data = input_stream.read(CHUNK)
            # 오디오 데이터 출력
            output_stream.write(data)
    except KeyboardInterrupt:
        print("Stream stopped.")

# 멀티쓰레딩을 사용하여 각 마이크에서 데이터를 읽고 각 스피커로 출력
threads = []
for i in range(len(INPUT_DEVICE_INDICES)):
    thread = threading.Thread(target=stream_audio, args=(inpu
    threads.append(thread)
    thread.start()

# 스레드 종료 대기
for thread in threads:
    thread.join()

# 스트림 닫기
for i in range(len(INPUT_DEVICE_INDICES)):
    input_streams[i].stop_stream()
    input_streams[i].close()
    output_streams[i].stop_stream()
    output_streams[i].close()

# 파이오디오 종료
p.terminate()
```

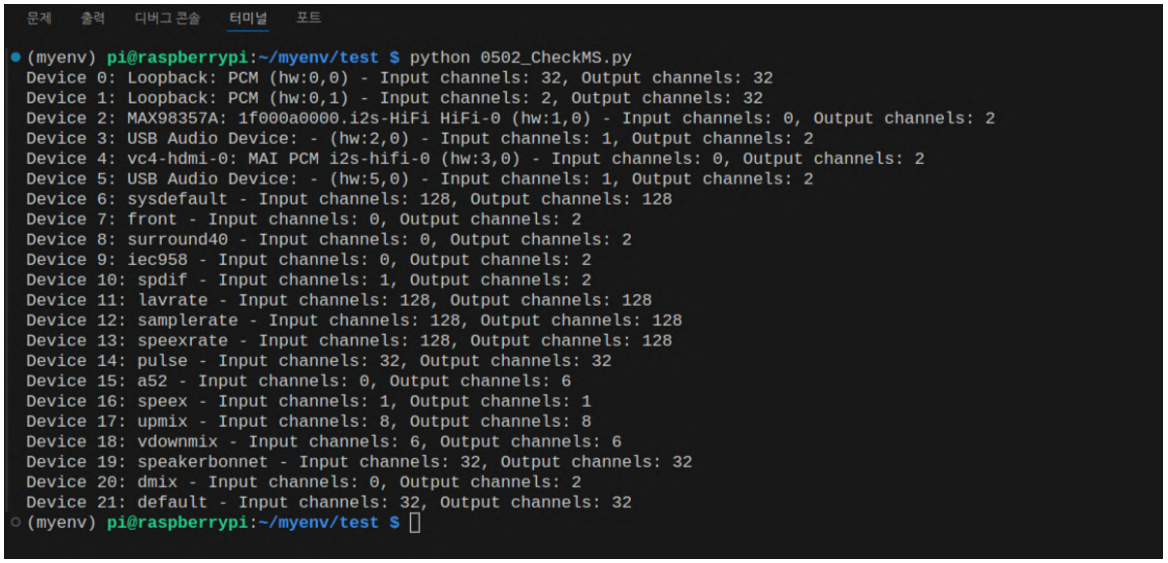
<최종 확인>

- 0502_CheckMS.py

```
import sounddevice as sd

# 장치 정보 출력
device_info = sd.query_devices()
```

```
for index, device in enumerate(device_info):
    print(f"Device {index}: {device['name']} - Input channels
```

→ 결과 화면



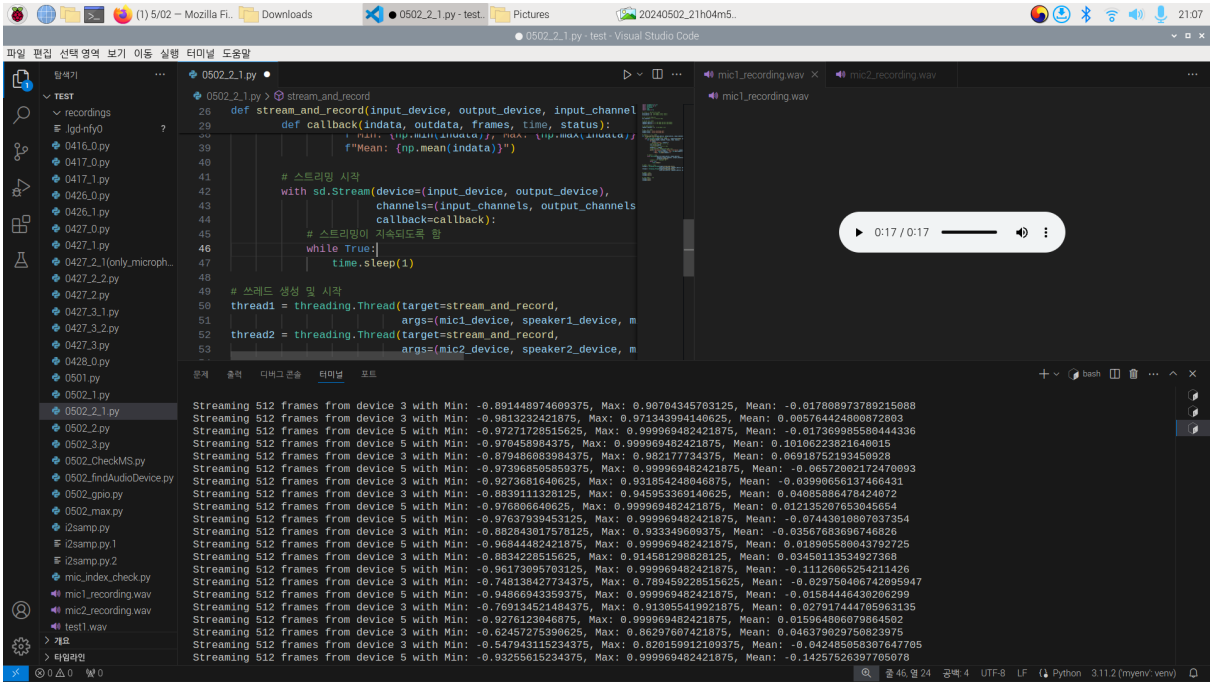
- 0502_2_1.py

```
import sounddevice as sd
import soundfile as sf
import threading
import time
import numpy as np

# 마이크 및 스피커 장치 설정
mic1_device = 3  # 첫 번째 마이크
mic2_device = 5  # 두 번째 마이크

# 각 마이크의 채널 수
mic_channels = 1  # 마이크는 모노

# 스피커 장치 설정
speaker1_device = 3  # 첫 번째 스피커
speaker2_device = 5  # 두 번째 스피커
```

```
# 스피커의 채널 수
speaker_channels = 2  # 스테레오 출력

# 녹음 파일 설정
output_file1 = "mic1_recording.wav"
output_file2 = "mic2_recording.wav"

# 스트리밍 및 녹음 함수
def stream_and_record(input_device, output_device, input_chan
    # WAV 파일 생성 및 쓰기 모드로 열기
    with sf.SoundFile(output_file, mode='w', samplerate=44100
        def callback(indata, outdata, frames, time, status):
            if status:
                print(f"Status: {status}")
            # 입력 데이터를 WAV 파일로 녹음
            file.write(indata)
            # 입력 데이터를 출력으로 복사
            outdata[:] = indata
            # 입력 데이터를 간략히 터미널에 출력
            print(f"Streaming {frames} frames from device {in
                  f"Min: {np.min(indata)}, Max: {np.max(indat
                  f"Mean: {np.mean(indata)}")

        # 스트리밍 시작
        with sd.Stream(device=(input_device, output_device),
                       channels=(input_channels, output_chann
                       callback=callback):
            # 스트리밍이 지속되도록 한다
            while True:
                time.sleep(1)

# 쓰레드 생성 및 시작
thread1 = threading.Thread(target=stream_and_record,
                           args=(mic1_device, speaker1_device
thread2 = threading.Thread(target=stream_and_record,
                           args=(mic2_device, speaker2_device
```

```
# 쓰레드 시작
thread1.start()
thread2.start()

# 메인 스레드 대기
thread1.join()
thread2.join()
```

→ 결과 화면



다음 할 일

- 스피커 연결하여 MAX98357a 증폭 모듈로 gpio제어 및 실제 출력 확인
- usb 마이크 연결하여 증폭 모듈 테스트 → 하나 되면 두 개 테스트

5 05/03 ~ 05/04 (1)

라즈베리파이 상에서 MAX98357a의 I2S프토토콜방식 문서를 찾을 수가 없어서, ESP32 보드를 가지고 테스트 진행 → 계속해서 오류 발생

하드웨어가 아닌 import os 를 하여 os.system 방식으로 sw적으로 소리 출력을 조절해보려 시도하였으나 오류 발생