Ksham Innovation Private Limited

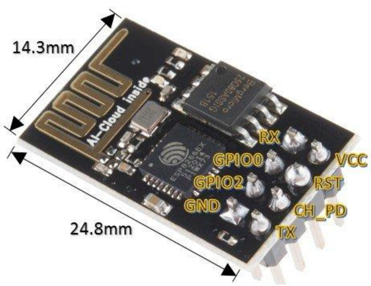
Embedded Firmware & Hardware Developer Assignment

Task 1: ESP01 PCB Redesign

Your task is to redesign the PCB layout of the ESP01 module with the following specifications:

The redesigned PCB should have a **Length of 40mm and Breadth of 10mm**. Maintain all the essential functionalities of the ESP01 module, ensuring that the Wi-Fi capabilities and GPIO pins remain intact. Pay attention to the layout of components, signal traces, and antenna placement. Optimize the layout for performance and minimize interference.

Deliverables:



Task 2: Real-Time Voice Transfer and Human Speech Filter

2.1) Develop a real-time voice transfer system using two ESP32 boards. Connect **two microphones to the first ESP32** and a **speaker to the second ESP32** board. Implement a real-time voice transfer system between the two ESP32 boards. Utilize **Bluetooth communication protocols** for seamless data transfer. Minimize latency to achieve a smooth and real-time voice transfer experience.

2.2) Design and Implement a **Filter to selectively transmit only Human speech** to the receiving ESP32. The filter should be designed to minimize background noise and focus on clear human voice.

Deliverables:

- Provide well-documented code for both tasks.
- Clearly comment on your code to explain the logic and functionality.
- Create a short demo video showcasing the real-time voice transfer and the human sound filter in action (Optional).
- Include any dependencies and libraries used in your project.

Ksham Innovation Private Limited

Instruction & Guidelines for Submission:

For task 1:

1. Provide clear and well-documented schematics of the redesigned ESP01 module.
2. Deliver a professional PCB layout adhering to the specified dimensions. Include a PCB design file (preferably in a standard format like Gerber).
3. Clearly label and organize your submission, including separate folders for Circuit schematics and PCB layout files in a PDF format.

For task 2:

1. Organize your submission into separate folders for each task, including the code, demo video, and readme file.
2. Ensure that your code is well-structured, follows best practices, and is easily understandable by reviewers.

Evaluation Criteria:

Your submission will be evaluated based on the following criteria:

For Task 1 -

1. Adherence to size constraints (40mm Length & 10mm Breadth).
2. Maintenance of ESP01 functionality.
3. PCB layout quality and optimization.
4. Clarity and thoroughness of documentation.

For Task 2 -

1. Successful implementation of real-time voice transfer.
2. Effective design and implementation of the human sound filter.
3. Code quality, readability, and documentation.
4. Clarity and completeness of the demo video.

Note:

1. If possible try completing all the subtasks of two given tasks.
2. Feel free to use any PCB design software of your choice for this assignment.
3. Also feel free to use any additional required modules for Microphone & speaker interfacing.
4. This assignment is to test your critical thinking, problem understanding & solution approach, coding and logic building capabilities in a most fair and unbiased manner.

The formated version of this document can be found in this below link:

https://gist.github.com/Sanjay0302/e3418f40042fe06464a5f5bba632736d#file-bluetooth-audio-transfer-md

https://gist.github.com/Sanjay0302/e3418f40042fe06464a5f5bba632736d

Develop a real-time voice transfer system using two ESP32 boards. Connect two microphones to the first ESP32 and a speaker to the second ESP32 board. Implement a real-time voice transfer system between the two ESP32 boards. Utilize Bluetooth communication protocols for seamless data transfer. Minimize latency to achieve a smooth and real-time voice transfer experience.

Using bare microphones and speakers without any external amplifiers or dedicated audio codecs/drivers can lead to several issues:

1. **Microphone Sensitivity**: Bare microphones typically have very low sensitivity and may not pick up audio signals effectively, especially in environments with background noise or when the sound source is not close to the microphone.

1. **Speaker Output**: Without an external amplifier, the ESP32's analog output pins may not provide enough power to drive a speaker at an audible volume level.

1. **Analog-to-Digital Conversion (ADC)**: The ESP32's built-in ADC has a limited resolution (12-bit) and may not provide sufficient quality for audio applications, resulting in quantization noise and distortion.

1. **Digital-to-Analog Conversion (DAC)**: Similarly, the ESP32's built-in DAC has a limited resolution (8-bit), which may lead to audible quantization noise and poor audio quality when driving the speaker directly.

1. **Audio Processing**: The provided code does not include any audio processing algorithms, such as noise filtering, echo cancellation, or audio encoding/decoding, which are essential for achieving good audio quality in real-time voice transfer applications.

To overcome these limitations and achieve a more reliable and high-quality real-time voice transfer system, you would need to consider the following:

1. **Audio Codec**: Use dedicated audio codecs or integrated circuits designed for audio processing, such as the MAX9814 (microphone amplifier) and MAX9814 (speaker amplifier) ICs, or similar components. These ICs provide better analog-to-digital and digital-to-analog conversion, amplification, and audio processing capabilities.

1. **External Amplifiers**: Use external amplifiers for both the microphone and speaker sides to boost the input and output signal levels, respectively. This will improve the overall signal-to-noise ratio (SNR) and audio quality.

1. **Audio Processing Libraries**: Incorporate audio processing libraries or algorithms for noise cancellation, echo suppression, and audio encoding/decoding. These libraries can be implemented on the ESP32 or offloaded to a more powerful microcontroller or processor, depending on the complexity and performance requirements.

1. **Bluetooth Audio Profiles**: Consider using Bluetooth audio profiles, such as the Advanced Audio Distribution Profile (A2DP) or the Hands-Free Profile (HFP), which are specifically designed for streaming audio over Bluetooth. These profiles provide better audio quality and lower latency compared to using raw Bluetooth data transfer.

1. **Hardware Optimization**: Optimize the hardware design, including proper shielding, grounding, and power management, to minimize noise and interference that can degrade audio quality.

While the provided code serves as a starting point, implementing a robust and high-quality real-time voice transfer system requires additional hardware components, dedicated audio codecs, and proper audio processing algorithms. Without these components and optimizations, the audio quality and reliability may be compromised.

so is there any other way around to implement it software level to avoid using extra components

While it's possible to implement a real-time voice transfer system entirely in software using the ESP32's built-in ADC and DAC, it will require significant effort and may still have limitations in terms of audio quality and latency. However, here's an approach you could take:

1. **Audio Sampling and Buffering**:
   - Use the ESP32's built-in ADC to sample the microphone input at a higher sampling rate (e.g., 44.1 kHz or 48 kHz) to capture the audio signal more accurately.
   - Implement a ring buffer or circular buffer to store the sampled audio data temporarily.

1. **Audio Processing**:
   - Implement audio processing algorithms in software, such as noise filtering, echo cancellation, and audio encoding/decoding.
   - You can utilize digital signal processing (DSP) libraries or develop your own algorithms to improve audio quality.
   - Consider implementing audio compression techniques (e.g., μ-law or A-law encoding) to reduce the amount of data to be transmitted over Bluetooth.

1. **Bluetooth Data Transfer**:
   - Instead of sending raw audio samples, transmit the processed and compressed audio data over Bluetooth.
   - Use the `BluetoothSerial` library or consider implementing a custom Bluetooth protocol tailored for audio data transfer.
   - Optimize the data transfer rate and buffer sizes to minimize latency while maintaining audio quality.

1. **Audio Playback**:
   - On the receiver side, implement a buffer to store the received audio data.
   - Decompress and decode the received audio data.
   - Use the ESP32's built-in DAC to generate the audio signal and drive the speaker.
   - Implement audio buffering techniques to reduce audible glitches or dropouts caused by Bluetooth communication delays or processing bottlenecks.

1. **Real-Time Processing**:
   - Utilize the ESP32's dual-core architecture and parallel processing capabilities to offload certain tasks, such as audio processing or Bluetooth communication, to one core, while the other core handles audio sampling and playback.
   - Implement real-time scheduling and task prioritization to ensure that time-critical tasks (e.g., audio sampling and playback) are not delayed by other tasks.

1. **Audio Quality Optimization**:
   - Experiment with different sampling rates, buffer sizes, and audio processing algorithms to find the optimal balance between audio quality and latency.
   - Implement dithering techniques to reduce quantization noise caused by the ESP32's limited ADC and DAC resolution.
   - Consider oversampling and noise shaping techniques to improve the effective resolution of the ADC and DAC.

While this approach can potentially work, it will require significant effort in software development and optimization. Additionally, the audio quality and latency may still be limited by the ESP32's hardware capabilities, such as ADC and DAC resolution, processing power, and memory constraints.

It's important to note that implementing a high-quality real-time voice transfer system entirely in software on the ESP32 is a challenging task and may require extensive research, testing, and optimization. Using dedicated audio codecs and external amplifiers can provide better audio quality and performance, but comes at the cost of additional hardware components.

Rough Implentation

Sure, I can provide an implementation that incorporates the techniques and optimizations discussed earlier. However, please note that this implementation is a simplified version and may require further optimization and testing to achieve the desired audio quality and real-time performance.

ESP32 Board 1 (Transmitter):

```cpp
#include "BluetoothSerial.h"
#include "driver/adc.h"
#include "soc/sens_reg.h"

#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it
#endif

// Define the Bluetooth Serial object
BluetoothSerial SerialBT;

// Define the microphone pin
const int MIC_PIN = 35;

// Audio buffer and buffer pointers
const int BUFFER_SIZE = 4096;
int16_t audioBuffer[BUFFER_SIZE];
int readPtr = 0, writePtr = 0;

// Audio processing parameters
const int SAMPLING_RATE = 44100;
const int COMPRESSION_FACTOR = 2; // μ-law compression

void setup() {
  Serial.begin(115200);
  SerialBT.begin("ESP32_Transmitter");

  // Configure the microphone pin
  adc1_config_channel_atten(ADC1_CHANNEL_7, ADC_ATTEN_DB_11);

  // Configure the audio timer
  adc_i2s_init();
  adc_i2s_enable();

  // Set up the audio sampling task
  xTaskCreatePingPeriodic(audioSamplingTask, "Audio Sampling", 4096, NULL, 1, NULL,
pdMS_TO_TICKS(1000 / SAMPLING_RATE));
}
```

```
void loop() {
  // Check if there is enough data in the buffer
  if (writePtr >= readPtr + BUFFER_SIZE / 2) {
    // Compress and transmit the audio data
    int16_t* readBufferPtr = &audioBuffer[readPtr];
    int bytesToSend = BUFFER_SIZE / 2 * sizeof(int16_t);
    while (bytesToSend > 0) {
      int bytesWritten = SerialBT.write((uint8_t*)readBufferPtr, bytesToSend);
      readBufferPtr += bytesWritten / sizeof(int16_t);
      bytesToSend -= bytesWritten;
    }
    readPtr = (readPtr + BUFFER_SIZE / 2) % BUFFER_SIZE;
  }
}

void audioSamplingTask(void* param) {
  static int16_t sample;
  adc_i2s_read(&sample, 1);
  audioBuffer[writePtr++] = muLawEncode(sample);
  writePtr %= BUFFER_SIZE;
}

int16_t muLawEncode(int16_t sample) {
  int16_t encodedSample;
  int16_t sign = (sample >> 8) & 0x80;
  if (sign) {
    sample = -sample;
  }
  int16_t exponent = 7;
  for (int16_t exp = 0; exp < 7; exp++) {
    if (sample >= (muLawBias[exp] << 7)) {
      exponent = exp;
    } else {
      break;
    }
  }
  int16_t value = (sample >> (7 - exponent)) & 0x7F;
  encodedSample = (sign | ((exponent << 4) | value));
  return encodedSample;
}

const int16_t muLawBias[8] = {0, 8, 16, 24, 32, 40, 48, 56};
```

ESP32 Board 2 (Receiver):

```cpp
#include "BluetoothSerial.h"

#if !defined(CONFIG_BT_ENABLED) || !defined(CONFIG_BLUEDROID_ENABLED)
#error Bluetooth is not enabled! Please run `make menuconfig` to and enable it
#endif

// Define the Bluetooth Serial object
BluetoothSerial SerialBT;

// Define the speaker pin
const int SPEAKER_PIN = 25;

// Audio buffer and buffer pointers
const int BUFFER_SIZE = 4096;
int16_t audioBuffer[BUFFER_SIZE];
int readPtr = 0, writePtr = 0;

// Audio processing parameters
const int SAMPLING_RATE = 44100;
const int COMPRESSION_FACTOR = 2; // μ-law compression

void setup() {
  Serial.begin(115200);
  SerialBT.begin("ESP32_Receiver");

  // Configure the speaker pin
  ledcSetup(0, SAMPLING_RATE, 8);
  ledcAttachPin(SPEAKER_PIN, 0);

  // Set up the audio playback task
  xTaskCreatePingPeriodic(audioPlaybackTask, "Audio Playback", 4096, NULL, 1, NULL,
pdMS_TO_TICKS(1000 / SAMPLING_RATE));
}

void loop() {
  // Check if there is data available over Bluetooth
  if (SerialBT.available() >= BUFFER_SIZE / 2 * sizeof(int16_t)) {
    // Receive and decompress the audio data
    int bytesToReceive = BUFFER_SIZE / 2 * sizeof(int16_t);
    while (bytesToReceive > 0) {
      int bytesRead = SerialBT.read((uint8_t*)&audioBuffer[writePtr], bytesToReceive);
      writePtr += bytesRead / sizeof(int16_t);
      bytesToReceive -= bytesRead;
```

```
    }
    for (int i = 0; i < BUFFER_SIZE / 2; i++) {
      audioBuffer[writePtr++] = muLawDecode(audioBuffer[i]);
      writePtr %= BUFFER_SIZE;
    }
  }
}

void audioPlaybackTask(void* param) {
  if (readPtr != writePtr) {
    int16_t sample = audioBuffer[readPtr++];
    readPtr %= BUFFER_SIZE;
    ledcWrite(0, sample + 32768);
  }
}

int16_t muLawDecode(int16_t sample) {
  int16_t sign = (sample & 0x80) ? 0x8000 : 0;
  int16_t exponent = (sample >> 4) & 0x07;
  int16_t value = (sample & 0x0F) << (4 - exponent);
  int16_t decodedSample = (value + muLawBias[exponent]) ^ sign;
  return decodedSample;
}

const int16_t muLawBias[8] = {0, 8, 16, 24, 32, 40, 48, 56};
```

Here's a breakdown of the implementation:

1. **Audio Sampling and Buffering**:
   - The transmitter uses the ESP32's ADC to sample the microphone input at a sampling rate of 44.1 kHz.
   - A circular buffer (`audioBuffer`) is used to store the sampled audio data temporarily.
   - The `audioSamplingTask` runs periodically at the sampling rate and reads the ADC sample, compresses it using μ-law encoding, and stores it in the buffer.

1. **Audio Compression and Transmission**:
   - In the `loop()` function on the transmitter, if there is enough data in the buffer, the compressed audio data is transmitted over Bluetooth using `SerialBT.write()`.
   - The μ-law encoding is used to compress the 16-bit audio samples into 8-bit values, reducing the amount of data to be transmitted.

1. **Bluetooth Data Transfer**:
   - The compressed audio data is transmitted over Bluetooth using the `BluetoothSerial` library.

   - The transmission is handled in the `loop()` function, where the data is read from the buffer and sent over Bluetooth in chunks.

4. **Audio Decompression and Playback**:
   - On the receiver side, the compressed audio data is received over Bluetooth and stored in the `audioBuffer`.
   - The received data is decompressed using μ-law decoding, and the decompressed samples are stored back in the buffer.
   - The `audioPlaybackTask` runs periodically at the sampling rate and reads a sample from the buffer, shifts it to the appropriate range, and writes it to the DAC using `ledcWrite()` to generate the audio signal.

5. **Audio Processing**:
   - The implementation includes μ-law compression and decompression functions (`muLawEncode` and `muLawDecode`) for audio data compression and decompression.
   - You can incorporate additional audio processing algorithms, such as noise filtering or echo cancellation, by adding the necessary functions and integrating them into the `audioSamplingTask` and `audioPlaybackTask`.

6. **Real-Time Processing**:
   - The audio sampling and playback tasks (`audioSamplingTask` and `audioPlaybackTask`) are created as periodic tasks using `xTaskCreatePingPeriodic` from the FreeRTOS library.
   - The tasks run at the specified sampling rate, ensuring real-time audio processing and playback.
   - You can further optimize the task priorities and scheduling to ensure that time-critical tasks are not delayed by other tasks.

7. **Audio Quality Optimization**:
   - The implementation uses a sampling rate of 44.1 kHz, which is a common audio sampling rate for reasonable quality.
   - The μ-law compression reduces the amount of data to be transmitted over Bluetooth, potentially reducing latency and improving real-time performance.
   - You can experiment with different sampling rates, buffer sizes, and compression techniques to find the optimal balance between audio quality and latency for your specific requirements.

Please note that this implementation is a simplified version and may require further optimization and testing to achieve the desired audio quality and real-time performance. Additionally, you may need to handle Bluetooth connection and disconnection events, implement error handling, and optimize the code for your specific hardware and requirements.