在 MA35D1 上实现实时语音对话的 ChatGPT

NuMicro® 64 位系列微控制器范例代码介绍

文件信息

应用简述	在 Nuvoton OpenWrt 平台上，利用 OpenAI Realtime API 实现支持语音输入与语音输出的实时响应型 ChatGPT，提供自然流畅的语音对话体验。
BSP 版本	MA35D1 Series OpenWrt 22.03
开发平台	NuMaker-IoT-MA35D16F90 V2.4

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. 概述

本范例程序展示了如何在 Nuvoton OpenWrt 嵌入式平台上，利用 Python 语言与 WebSocket 技术，结合 OpenAI 最新推出的 Realtime API，实现即时语音交互式对话系统。用户通过麦克风说话后，设备会实时录音并上传音频，OpenAI 模型则快速回复语音内容，并由系统即时播放，使人机交互更加自然、快速、直观。

这种低延迟交互能力，使嵌入式平台也具备构建智能语音助手、语音客服、教育对话机器人等场景的潜力。即使在资源受限的平台上，通过云端 AI 能力，仍可提供媲美手机等高端设备的语音交互体验。

本范例程序基于 Nuvoton OpenWrt 22.03 环境开发，需先搭建 Nuvoton OpenWrt 22.03 开发环境（https://github.com/OpenNuvoton/Nuvoton-OpenWrt-22.03），才能运行本范例程序。

虽然本文以 NuMaker-IoT-MA35D16F90 V2.4 开发板为例，但内容同样适用于所有支持 Nuvoton OpenWrt 22.03 且可流畅运行 Python 程序的开发板，包括 MA35D0、MA35D1 和 MA35H0 等系列。

1.1 原理

OpenAI Realtime API 是 OpenAI 为实时语音交互场景设计的新一代流式接口，使开发者能够通过语音输入和语音输出的方式与 GPT-4o 模型进行实时对话。它采用 WebSocket / WebRTC 双向通信机制，将语音识别（ASR）、语义理解（LLM）和语音合成（TTS）整合在同一连接中，实现真正的低延迟语音交互体验。

相比传统以 HTTP 为主的 Chat API，Realtime API 支持持续流式音频传输，用户在说话的同时即可开始识别和理解，无需等待完整语句输入完毕，能够有效将响应时间缩短到数百毫秒级别。

本范例程序的核心原理是结合本地语音设备（arecord / aplay），使用 Python 配合 WebSocket 技术，连接 OpenAI Realtime API 与 OpenAI 服务器，实现双向实时语音对话。整体流程如下：

1. 语音输入处理：通过 Linux 的 arecord 工具进行录音，采集语音数据。
2. 音频流上传：录音数据经 Base64 编码后，实时发送至 OpenAI Realtime API（gpt-4o-realtime-preview）。

3. 语音识别与理解：OpenAI 服务器根据流式语音进行实时语音识别，并根据识别结果返回语音回复和文字转录。

4. 语音回复处理：接收到的语音回复（Base64 编码的语音片段）解码后通过 aplay 播放输出。

5. 多轮对话支持：程序保留用户提问与 GPT 回复内容，以维持对话上下文，提升交互连贯性。

程序中利用 Python websockets 库建立与 OpenAI 服务器的双向 WebSocket 连接，并采用 asyncio 架构设计「录音发送」、「回复接收」和「音频播放」三个异步任务，确保全流程的非阻塞处理与实时响应。

此外，为了更稳定的语音检测，范例程序采用服务器端语音起止检测（server_vad），当检测到语音停止时自动终止录音。这样不仅无需按键操作来控制录音的开始和结束，还能节省资源并提升交互效率。

相比传统通过文字发送请求的 ChatGPT 交互模式，本范例程序以语音为主的方式更贴近自然语言交流习惯，也适用于无屏幕或免触控设备，使嵌入式系统也能提供现代语音 AI 体验。

1.2 环境设定

本范例程序采用 Python 语言编写，运行过程中会使用多个 Python 模块和 Linux 工具。因此，在执行前必须先确认这些相关模块和工具均已安装并可正常运行。由于 OpenWrt 系统本身能够方便地支持这些必要工具，因此本说明以 OpenWrt 作为开发与运行环境。

关于 MA35D1 平台的构建流程及镜像文件生成方式，可参考 Nuvoton OpenWrt 22.03 Project User Manual。

- **OpenWrt 设置步骤**

在 OpenWrt 系统中，请按照以下步骤在 menuconfig 中启用本范例所需的软件包：

1. 执行 make menuconfig 命令以打开 OpenWrt 软件包菜单。可通过方向键选择分类，并按空格键启用软件包。也可以按「/」键并输入软件包名称进行快速搜索。

2. 本范例程序采用 Python 编写，并通过 WebSocket 与 OpenAI Realtime API 建立连接，因此需启用以下与 Python 相关的软件包：

- Languages ---> Python ---> <*> python3..... Python 3.10 programming language
- Languages ---> Python ---> <*> python3-websockets.. Library for developing WebSocket servers and clients

3. 语音输入与输出部分使用 arecord 和 aplay 工具，需要启用 ALSA 音频支持相关软件包：

   - Sound ---> <*> alsa-utils..... ALSA (Advanced Linux Sound Architecture) utilities

4. 启用所有必要软件包后，请退出 menuconfig 并保存设置。

5. 将 *chatgpt_audio.py* 文件拷贝到 *target/linux/ma35d1/base-files/* 目录，使其在系统构建完成后可直接位于 OpenWrt 的根目录下。

6. 执行 make 命令进行系统构建，使用更新后的设置生成镜像文件。

构建完成后，可通过 NuWriter 将镜像文件烧录到 NuMaker-IoT-MA35D16F90 开发板中，完成环境部署。

1.3 执行结果

本范例程序的 OpenWrt 映像文件执行编译结果如图 1-1 所示。编译完成后，可以在 *bin/targets/ma35d1/iot/* 目录下找到生成的映像文件。

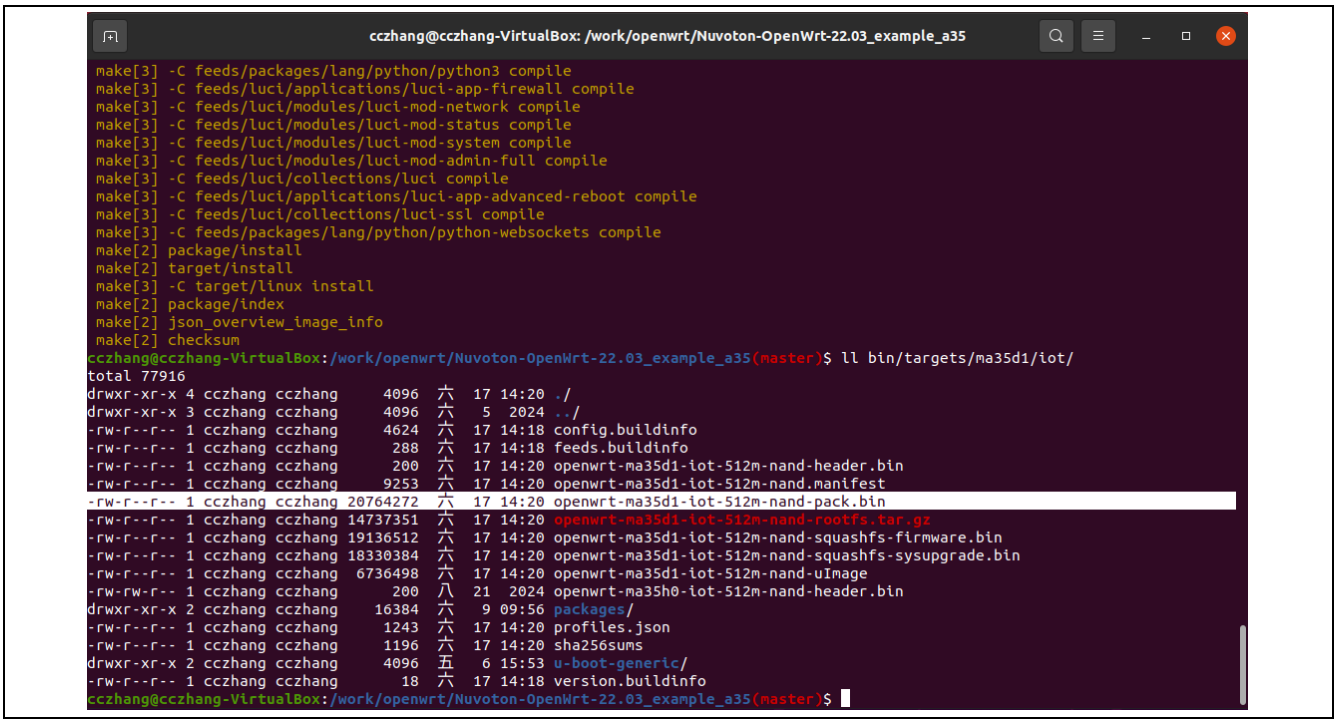

图 1-1 OpenWrt 执行编译结果

本范例程序的 OpenWrt 映像文件执行烧录结果，如图 1-2 所示。

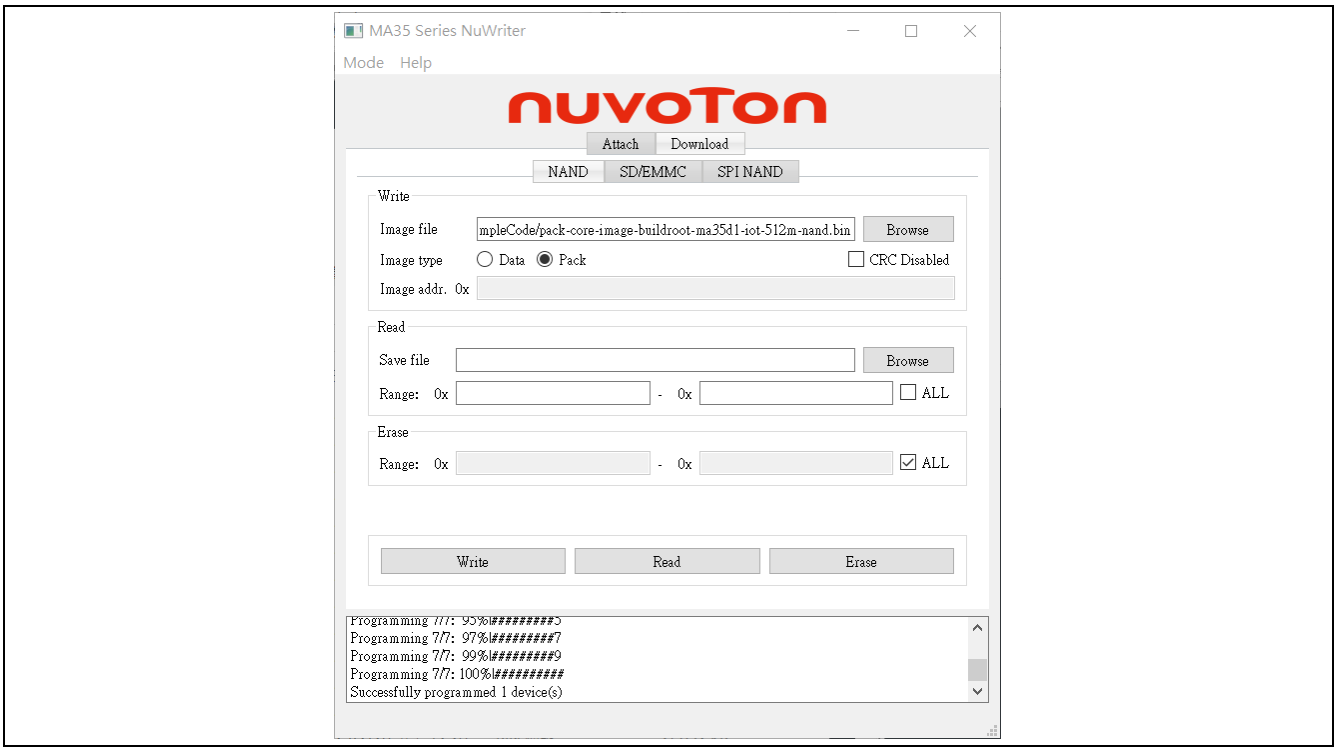

图 1-2 使用 NuWriter 烧录 OpenWrt 映像文件

本范例程序的执行流程与结果如下。当程序启动后，界面上会显示「== Please speak into the microphone... ==」提示，用户即可开始对着麦克风进行语音输入。系统采用语音活动检测（VAD）来判断语音的开始与结束，根据语音与静音的间隔进行判定。因此，如果想连续说出多个句子，中间只能停顿很短的时间，否则会被视为语音输入已完成。

当界面显示「>> Speech started...」时，表示服务器已检测到语音输入开始；而出现「<< Speech stopped...」时，则代表服务器判定语音输入已结束。接着，服务器会返回对应的语音和文字回复内容。文字通常会在短时间内全部显示完毕，而语音则需要依次播放每段音频，播放过程无法加速，因此整体所需时间较长。当界面显示「Response done for <...>」时，代表本次提问的回复已生成完毕，但仍需等语音播放结束后，才能进行下一轮提问。

以下为实际执行时的终端输出信息：

```
root@OpenWrt:/# python chatgpt_audio.py

==  Please speak into the microphone...  ==
2025-06-18 08:01:22,494 [INFO] >> Speech started...
2025-06-18 08:01:25,217 [INFO] << Speech stopped...
地球上的海水大约有13.32亿立方公里，占地球表面的70%左右。也就是说，海洋几乎覆盖了地球表面的绝大部分。这些海水如果均匀铺开来，可以形成一层大约3.7公里深的水层。
```

```
2025-06-18 08:01:31,049 [INFO] Response done for <在地球上有多少海水?>
```

本范例程序在终端软件中执行语音聊天功能时的输出结果，这里以 PuTTY 为例，如图 1-3 所示。gpt-4o-realtime-preview 模型具备自动语言识别能力，能够支持多语言混合输入以及不同口音的识别。因此，用户在与 ChatGPT 互动过程中，可以自由切换语言，例如中文、英文或日文，系统都能正确识别并给出相应语言的回复。

如图所示，用户可以依次用不同语言提问，系统会实时显示语音输入、识别、回复等过程，并根据输入语言自动切换回复内容。这大大提升了跨语言交流的灵活性与便利性，适合多语言环境下的应用需求。

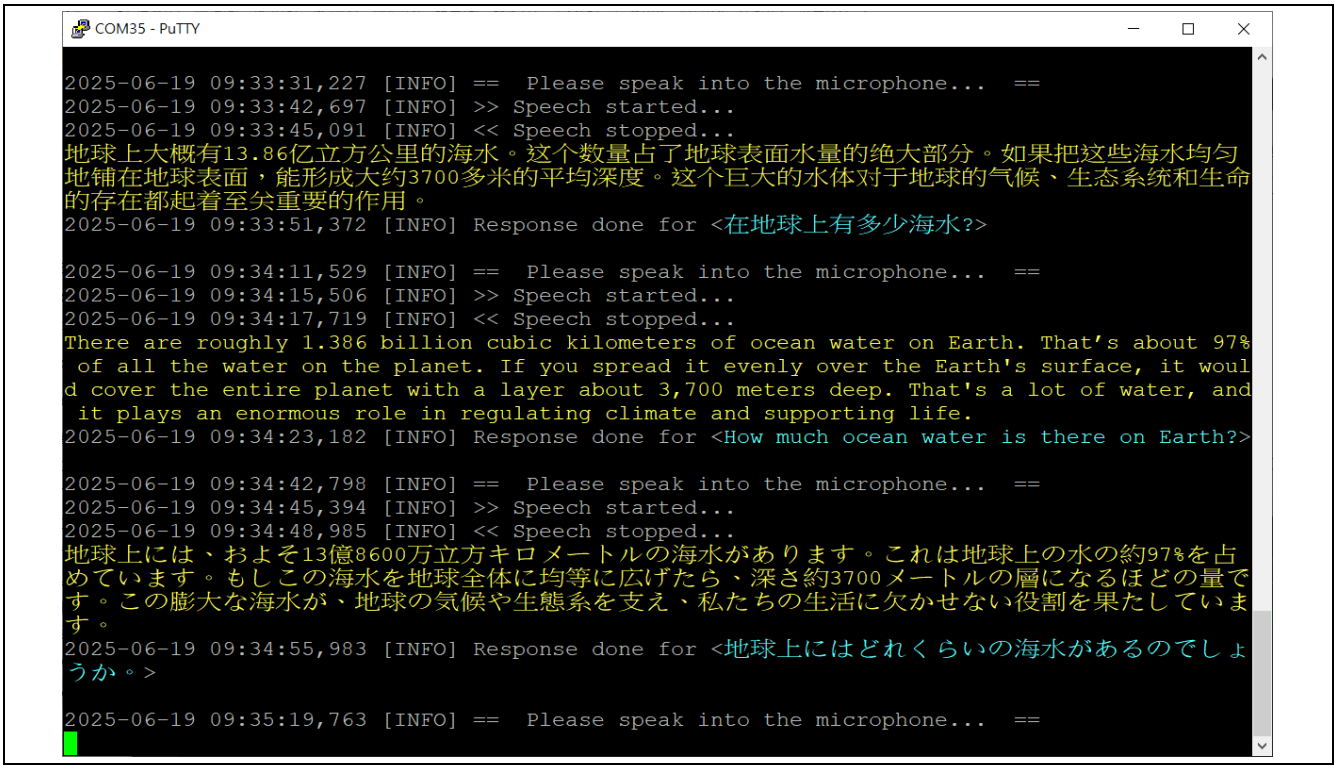

图 1-3 ChatGPT 聊天功能示例 1

gpt-4o-realtime-preview 模型具备强大的上下文记忆能力，能够支持多轮对话。这意味着每次模型生成回复时，不仅会考虑用户刚刚输入的内容，还会参考整个对话过程中的历史记录，进行语义判断和回复生成。用户可以连续提出相关问题，系统会根据上下文给出更加自然且连贯的回答。

如图 1-4 所示，用户可以多次与 ChatGPT 互动，针对之前的对话内容进行追问或延伸讨论，gpt-4o-realtime-preview 模型都能正确理解并延续对话，展现多轮交互的能力。

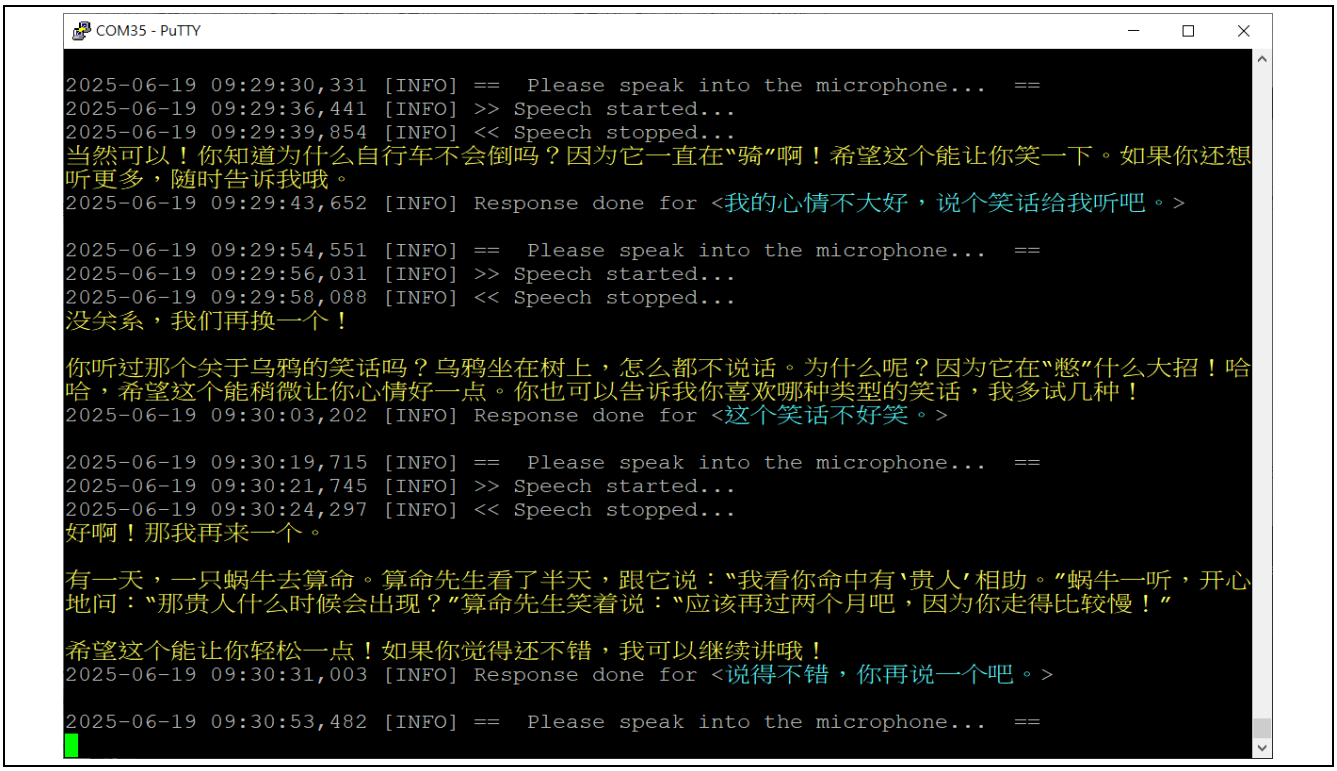

图 1-4 ChatGPT 聊天功能示例 2

gpt-4o-realtime-preview 模型不仅能够进行日常对话，也具备处理复杂条件和展现创意的能力。只要用户给出明确的目标或指令，模型就能根据需求生成结构完整且内容丰富的回复。这使得该系统能够应用于故事创作、规划建议、条目分析等多种场景。

如图 1-5 所示，用户可以针对特定主题或条件提出请求，gpt-4o-realtime-preview 模型会根据指令自动生成详细且富有创意的内容，展现出强大的语言生成能力。

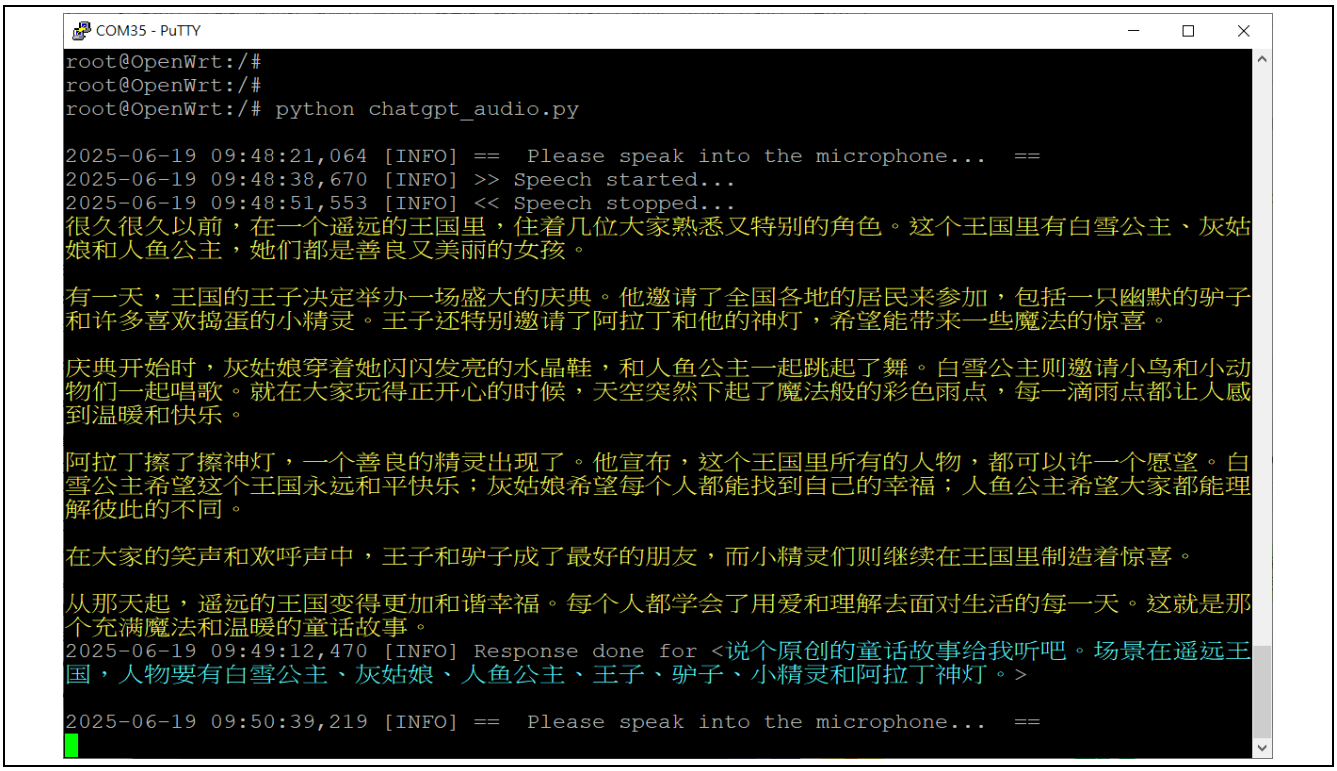

图 1-5 ChatGPT 聊天功能示例 3

2. 代码介绍

在程序初始化阶段，通过 logging.basicConfig() 设置日志输出格式和级别。默认级别为 INFO，可以显示基本的运行信息；如果将级别设置为 DEBUG，则可以输出更详细的调试信息，包括异步任务状态、事件内容等，有助于开发和问题追踪。此外，DEBUG 模式还会显示其他模块（如 websockets、asyncio）的内部信息，便于观察整个系统的运行情况。

```
# Configure logging: set level to INFO or DEBUG as needed
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(message)s"
)
```

AudioStreamer 类是整个语音交互流程的核心，负责处理音频流、WebSocket 传输、语音播放以及对话状态管理。类在创建时会初始化多个属性，其中 question 用于存储用户语音输入最终识别出的提问文本；full_transcript 则用于累加来自服务器的语音识别增量回复，以重建完整的语音输入内容。ws 为 WebSocket 连接对象，负责与 OpenAI Realtime API 建立双向通信；_arecord_proc 用于控制录音子进程；audio_play_queue 则是异步队列，负责将语音回复数据传递到播放流程。

```
class AudioStreamer:
    def __init__(self) -> None:
        self.question = ""              # The user's recognized question text
        self.full_transcript = ""       # The full transcript buffer (incremental)
        self.ws = None                  # WebSocket connection, set after connect
        self._arecord_proc = None       # arecord process, set when recording starts
        self.audio_play_queue = None  # Will be set to a new Queue in run_loop
```

当 WebSocket 连接建立完成后，update_session() 方法会向 OpenAI Realtime API 发送一条「session.update」消息，用于设置语音转文字模型和语音起止检测（turn detection）的参数。本范例指定使用 gpt-4o-transcribe 模型进行语音识别，并启用服务器端的语音活动检测（Server VAD），使系统能够自动判断用户何时开始和结束说话，无需在设备端额外处理启停逻辑，从而提升语音交互的实时性和稳定性。

其中 turn detection 参数区块的各个变量说明如下：

- type：设置为 server_vad，启用服务器端语音起止判断机制，由服务器自动检测语音输入区段。

- threshold：语音检测的灵敏度阈值，范围为 0.0 到 1.0。数值越低（如 0.3），越容易判断为开始说话，适合安静环境；数值越高（如 0.7），可减少背景噪音误触，适合嘈杂环境。本范例设置为 0.5，属于中等灵敏度。
- prefix_padding_ms：当服务器检测到语音开始时，回补指定毫秒数的音频作为语句开头，防止语音被截断。本范例设置为 800 毫秒，适合一般语速；语速较慢的用户可尝试增加到 1000 毫秒。
- silence_duration_ms：当检测到用户持续静音超过该毫秒数后，即视为语音输入结束。本范例设置为 500 毫秒，以加快响应速度；缩短到 300 毫秒可进一步提升速度，但可能导致语音过早截断。

```
async def update_session(self) -> None:
    """
    Send a session update message to the server to configure
    audio transcription and turn detection parameters.
    """
    if self.ws:
        await self.ws.send(json.dumps({
            "type": "session.update",
            "session": {
                "input_audio_transcription": {"model": "gpt-4o-transcribe"},
                "turn_detection": {
                    "type": "server_vad",
                    "threshold": 0.5,           # VAD sensitivity threshold
                    "prefix_padding_ms": 800,   # Audio before speech start to include
(ms)
                    "silence_duration_ms": 500, # Silence duration to consider speech
ended (ms)
                }
            }
        }))
```

audio_send_worker() 方法负责启动录音程序 arecord，并将录制的语音数据实时流式传送到 OpenAI Realtime API。该方法在执行时会启动一个子进程，从标准输出读取录音数据，每次读取一小段音频后，会先将数据转换为 Base64 编码，再通过 WebSocket 发送到服务器，事件类型为「input_audio_buffer.append」。该进程会持续运行，直到服务器端的语音活动检测（VAD）判定用户已说完话，并由其他协同任务主动终止录音子进程。整体流程确保用户的语音输入在说话过程中能够稳定且实时地传送到服务器，为语音识别和后续回复提供即时且连续的数据来源。

```
async def audio_send_worker(self) -> None:
    """
    Start arecord to capture microphone audio and stream audio chunks
    to the websocket in real time.
    """
```

```
    logging.debug("@@ audio_send_worker started")
    cmd = [
        "arecord",
        "-f", "S16_LE",      # 16-bit little-endian PCM
        "-r", "24000",       # 24 kHz sample rate
        "-c", "1",           # Mono channel
    ]
    # Start arecord process with stdout as PIPE for streaming audio
    if logging.getLogger().getEffectiveLevel() == logging.DEBUG:
        proc = await asyncio.create_subprocess_exec(*cmd, stdout=asyncio.subprocess.PIPE)
    else:
        proc = await asyncio.create_subprocess_exec(
            *cmd,
            stdout=asyncio.subprocess.PIPE,
            stderr=subprocess.DEVNULL  # Suppress arecord stderr unless debugging
        )
    self._arecord_proc = proc  # Save process reference for later termination

    while True:
        # Read a chunk of audio data from arecord
        audio_chunk = await proc.stdout.read(4096)
        if not audio_chunk:
            break
        # Encode audio chunk as base64 and send to websocket
        payload = base64.b64encode(audio_chunk).decode("utf-8")
        await self.ws.send(json.dumps({
            "type": "input_audio_buffer.append",
            "audio": payload,
        }))
```

audio_play_worker() 方法负责播放模型返回的语音回复数据，具体实现上会启动 aplay 播放程序，并通过异步队列 audio_play_queue 持续接收音频数据。该方法会等待队列中出现新的语音片段，每获取一条数据后便立即写入 aplay 的输入管道进行播放。播放过程会持续进行，直到队列中收到 None 作为终止信号，表示本轮语音数据的传送已完成。此时程序会关闭输入管道，并等待 aplay 播放完最后一条音频数据后再结束进程，以确保语音内容完整播放且不会被截断。

```
async def audio_play_worker(self) -> None:
    """
    Play audio data from audio_play_queue using aplay.
    Terminates when None is received from the queue.
    """
    logging.debug("@@ audio_play_worker started")
    cmd = [
        "aplay",
        "-f", "S16_LE",      # 16-bit little-endian PCM
        "-r", "24000",       # 24 kHz sample rate
        "-c", "1",           # Mono channel
        "-"
    ]
    # Start aplay process for audio playback
    if logging.getLogger().getEffectiveLevel() == logging.DEBUG:
        proc = await asyncio.create_subprocess_exec(*cmd, stdin=asyncio.subprocess.PIPE)
```

```
    else:
        proc = await asyncio.create_subprocess_exec(
            *cmd,
            stdin=asyncio.subprocess.PIPE,
            stdout=subprocess.DEVNULL,  # Suppress aplay output unless debugging
            stderr=subprocess.DEVNULL
        )
    while True:
        # Wait for audio data from the queue
        audio = await self.audio_play_queue.get()
        if audio is None:
            break  # None signals end of playback
        proc.stdin.write(audio)
        await proc.stdin.drain()
    proc.stdin.close()      # Close aplay's stdin to signal end of audio stream
    await proc.wait()       # Wait for aplay process to finish playing all audio
```

ws_receive_worker() 方法负责接收并处理来自 OpenAI Realtime API 的各类 WebSocket 消息，包括语音检测、语音识别、语音回复和交互结束等事件。该方法通过异步循环不断监听并解析服务器返回的消息，并根据不同的事件类型进行相应的处理流程。

当接收到以下事件时，程序会做出如下操作：

- session.created：表示 WebSocket 会话已成功建立，系统可以开始传送语音数据。
- input_audio_buffer.speech_started：服务器检测到用户开始说话，系统会在终端输出提示，显示语音录制已启动。
- input_audio_buffer.speech_stopped：服务器检测到用户停止说话，此时会立即终止录音子进程（arecord），释放系统资源。
- response.audio_transcript.delta：模型持续返回的语音识别增量文本会被拼接到 full_transcript，并实时显示在界面上。
- response.audio.delta：返回模型语音回复数据的片段，系统会解码并送入播放队列，由播放任务处理输出。
- response.done：表示本轮语音交互已结束，系统会发送终止信号（None）到播放队列，结束本次回复流程。
- conversation.item.input_audio_transcription.completed：返回整段语音输入的最终识别结果，该文本会被追加到 question，作为本轮提问内容。

此外，如果 WebSocket 连接被中断（例如服务器关闭或网络中断），系统会捕获异常，自动结束该接收任务，避免整个应用程序异常终止。

总体来说，该方法是语音交互系统的接收核心，负责实时处理和驱动对话流程中的各项关键事件，确保整体交互行为具备完整性、实时性和同步性。

```
async def ws_receive_worker(self) -> None:
    """
    Receive and process messages from the WebSocket server.
    Handles audio, transcript, and session events.
    """
    logging.debug("@@ ws_receive_worker started")
    try:
        async for m in self.ws:
            evt = json.loads(m)
            # Avoid excessive log output for response.audio.delta
            if evt["type"] == "response.audio.delta":
                logging.debug(f"** Received: {m[:100]}...")
            else:
                logging.debug(f"** Received: {m}")

            if evt["type"] == "session.created":
                # Session is ready, can start sending audio
                logging.debug("Session created, ready to send audio")

            elif evt["type"] == "input_audio_buffer.speech_started":
                # Server detected speech start
                logging.info(">> Speech started...")

            elif evt["type"] == "input_audio_buffer.speech_stopped":
                # Server detected speech end
                logging.info("<< Speech stopped...")
                # Terminate arecord process when speech is stopped
                if self._arecord_proc:
                    self._arecord_proc.terminate()
                    try:
                        await self._arecord_proc.wait()
                    except Exception:
                        pass
                    self._arecord_proc = None

            elif evt["type"] == "response.audio_transcript.delta":
                # Append transcript delta to full transcript buffer
                delta_content = evt.get("delta", "")
                self.full_transcript += delta_content
                if logging.getLogger().getEffectiveLevel() >= logging.INFO:
                    print(f"\033[93m{delta_content}\033[0m", end='', flush=True)

            elif evt["type"] == "response.audio.delta":
                # Received audio response chunk, decode and queue for playback
                audio = base64.b64decode(evt["delta"])
                await self.audio_play_queue.put(audio)

            elif evt["type"] == "response.done":
                # End of response, signal playback worker to stop
                print()
                q = self.question.strip()
```

```
                logging.info(f"Response done for <\033[96m{q}\033[0m>")
                await self.audio_play_queue.put(None)
                break

            elif evt["type"] == "conversation.item.input_audio_transcription.completed":
                # Final transcript for the user's input audio
                transcript_content = evt.get("transcript", "")
                self.question += transcript_content

    except websockets.exceptions.ConnectionClosed:
        # WebSocket connection closed, exit receive worker
        logging.warning("** WebSocket closed, receive worker exiting")
```

run_loop() 方法是整个程序的主要入口，负责建立和管理 WebSocket 连接，初始化异步工作流程（录音、接收、播放），并处理回合控制与错误恢复。程序会进入一个永久循环，尝试与 OpenAI Realtime API 建立 WebSocket 连接，一旦连接成功，便调用 update_session()方法发送初始设置，并准备进行一次语音交互。

每一轮交互开始时，会重置输出队列 audio_play_queue，并启动三个异步任务：audio_send_worker() 负责发送语音数据，ws_receive_worker() 接收服务器响应事件，audio_play_worker() 播放语音回复。这些任务会并行执行，直到接收到服务器的结束信号为止。

交互结束后，程序会等待各任务完成，并发送终止信号以结束播放流程。如果连接中途断开，将进入异常处理块，稍作延迟后自动重新连接。最后，通过 finally 块完成录音和播放进程的清理、任务取消、队列释放与连接关闭，确保系统稳定并能重新启动下一轮交互。

```
async def run_loop(self) -> None:
    """
    Main entry point: setup WebSocket, start workers, and manage session.
    Handles auto-reconnect and session management.
    """
    while True:
        play_task = send_task = receive_task = keepalive_task = None
        self.audio_play_queue = None  # Reset queue reference at the start of each
connection
        try:
            # Connect to OpenAI Realtime WebSocket API
            self.ws = await websockets.connect(
                uri="wss://api.openai.com/v1/realtime?model=gpt-4o-realtime-preview-2025-
06-03",
                extra_headers={
                    "Authorization": f"Bearer {os.getenv('OPENAI_KEY')}",
                    "OpenAI-Beta": "realtime=v1",
                },
                ping_interval=10,   # Send ping every 10 seconds
                ping_timeout=5,     # If no pong in 5 seconds, consider disconnected
            )
            await self.update_session()
```

```
            while True:
                self.reset_buffer()
                # Create a new audio_play_queue for each round, bound to the current event
loop
                self.audio_play_queue = asyncio.Queue()
                play_task = asyncio.create_task(self.audio_play_worker())
                send_task = asyncio.create_task(self.audio_send_worker())
                receive_task = asyncio.create_task(self.ws_receive_worker())

                print()
                logging.info("==  Please speak into the microphone...  ==")

                # Wait for all workers to finish
                await receive_task
                await send_task
                await self.audio_play_queue.put(None)
                await play_task
                # Cancel keepalive task after each round if needed
                if keepalive_task and not keepalive_task.done():
                    keepalive_task.cancel()
                    try:
                        await keepalive_task
                    except asyncio.CancelledError:
                        pass
        except websockets.exceptions.ConnectionClosed as e:
            # WebSocket closed, try to reconnect after a short delay
            logging.warning(f"WebSocket closed: {e}. Reconnecting in 2s...")
            await asyncio.sleep(2)
        except Exception as e:
            # Log any unexpected errors and print traceback
            logging.error(f"!! Program error: {e}")
            traceback.print_exc()
            await asyncio.sleep(2)
        finally:
            # Force kill any arecord/aplay process (for MA35D1 OpenWrt stability)
            if self._arecord_proc:
                self._arecord_proc.terminate()
                try:
                    await self._arecord_proc.wait()
                except Exception:
                    pass
                self._arecord_proc = None
            # Kill any lingering arecord/aplay processes (may affect other users)
            os.system("killall -9 arecord aplay >/dev/null 2>&1")
            # Cancel all running tasks if they exist and are not done
            for t in (play_task, send_task, receive_task, keepalive_task):
                if t is not None and not t.done():
                    t.cancel()
                    try:
                        await t
                    except asyncio.CancelledError:
                        pass
            # Ensure playback worker is unblocked if waiting
            if self.audio_play_queue is not None:
                try:
                    self.audio_play_queue.put_nowait(None)
```

```
                except Exception:
                    pass
            # Close websocket connection if open
            if self.ws:
                await self.ws.close()
```

main() 方法是程序的启动入口，会先检查环境变量 OPENAI_KEY 是否存在，如果未设置则显示错误并结束程序。如果检查通过，则创建 AudioStreamer 对象并执行 run_loop() 方法，启动语音交互流程。整个流程被包含在 try-except 块中，如果发生异常错误，将输出错误信息和堆栈追踪，方便后续调试。

```
def main() -> None:
    """
    Entry point: check environment, create AudioStreamer, and run the main loop.
    """
    openai_key = os.environ.get('OPENAI_KEY')
    if not openai_key:
        logging.error("!! Please set the OPENAI_KEY environment variable")
        sys.exit(1)
    try:
        streamer = AudioStreamer()
        asyncio.run(streamer.run_loop())
    except Exception as e:
        logging.error(f"!! Program error: {e}")
        traceback.print_exc()

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

3. 软件与硬件需求

3.1 软件需求

- BSP 版本
  - MA35D1 Series OpenWrt 22.03

3.2 硬件需求

- 电路组件
  - NuMaker-IoT-MA35D16F90 V2.4

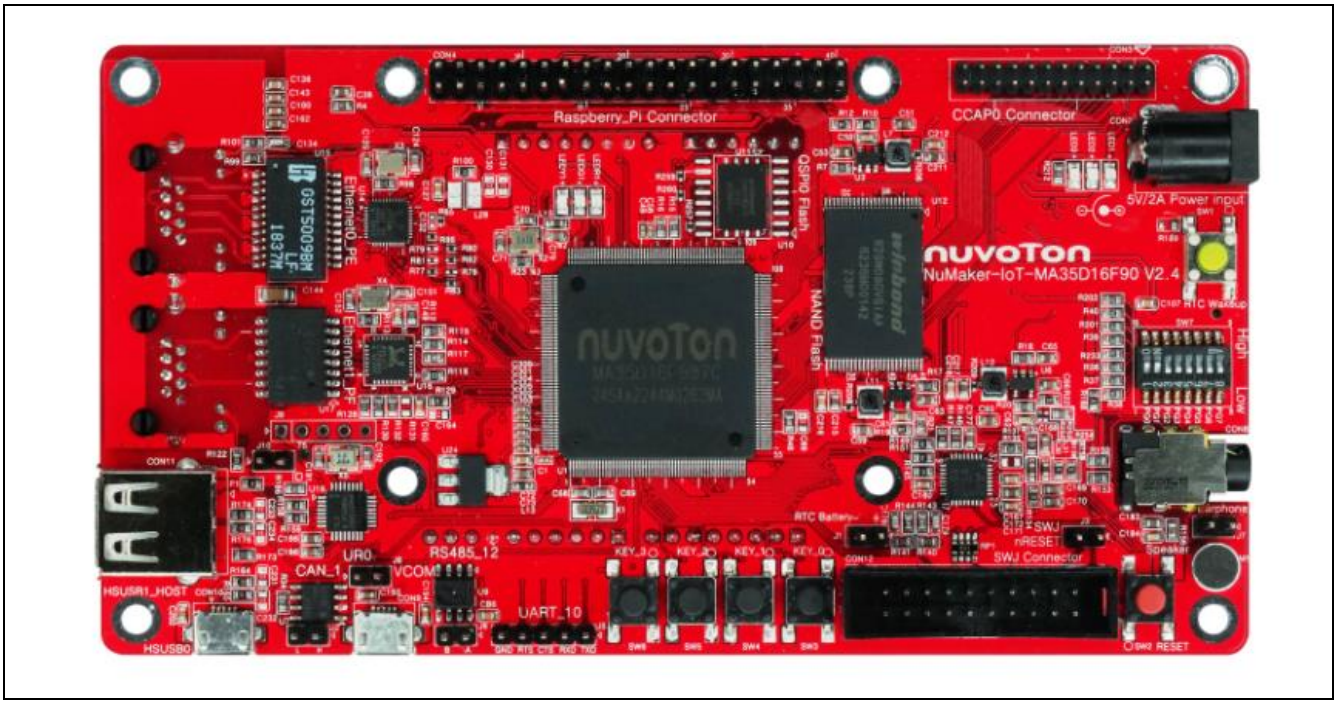

图 3-1 NuMaker-IoT-MA35D16F90 V2.4

- 语音设备
  - TRRS 接口耳机麦克风

4. 范例程序执行

1. 使用终端软件连接 NuMaker-IoT-MA35D16F90 开发板的 VCOM。

2. 将带有 TRRS 接口的耳机麦克风插入开发板。

3. 系统启动后，请确认设备已成功连接网络，且系统时间正确，这是正常运行 ChatGPT 范例程序的前提。可以使用 ping 和 date 命令进行确认：

```
root@OpenWrt:/# ping -c 1 api.openai.com
PING api.openai.com (162.159.140.245): 56 data bytes
64 bytes from 162.159.140.245: seq=0 ttl=54 time=76.843 ms

--- api.openai.com ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 76.843/76.843/76.843 ms
root@OpenWrt:/# date
Fri Jun 20 01:59:19 UTC 2025
```

4. 执行前可以先调整耳机的音量到合适的大小：

```
root@OpenWrt:/# amixer set Headphone 30
```

5. 设置 OpenAI API 密钥环境变量（请将 your_key_here 替换为实际密钥）：

```
root@OpenWrt:/# export OPENAI_KEY=your_key_here
```

6. 运行范例程序，即会进入语音对话模式。当界面出现提示信息时，即可开始语音输入，流程如前述章节所示：

```
root@OpenWrt:/# python chatgpt_audio.py

2025-06-19 09:51:18,276 [INFO] ==  Please speak into the microphone...  ==
```

7. 如需调试，可以将程序中的 logging 等级从 INFO 改为 DEBUG。启用后，运行时将显示更详细的调试信息，方便观察连接和事件流程：

```
root@OpenWrt:/# python chatgpt_audio.py
2025-06-20 02:49:18,191 [DEBUG] Using selector: EpollSelector
2025-06-20 02:49:19,101 [DEBUG] = connection is CONNECTING
2025-06-20 02:49:19,268 [DEBUG] > GET /v1/realtime?model=gpt-4o-realtime-preview-2025-06-
03 HTTP/1.1
2025-06-20 02:49:19,270 [DEBUG] > Host: api.openai.com
2025-06-20 02:49:19,272 [DEBUG] > Upgrade: websocket
2025-06-20 02:49:19,273 [DEBUG] > Connection: Upgrade
2025-06-20 02:49:19,274 [DEBUG] > Sec-WebSocket-Key: FOAupXkQjvDg7fdkdFmlLQ==
2025-06-20 02:49:19,275 [DEBUG] > Sec-WebSocket-Version: 13
2025-06-20 02:49:19,276 [DEBUG] > Sec-WebSocket-Extensions: permessage-deflate;
client_max_window_bits
```

```
2025-06-20 02:49:19,277 [DEBUG] > Authorization: Bearer sk-
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
2025-06-20 02:49:19,278 [DEBUG] > OpenAI-Beta: realtime=v1
2025-06-20 02:49:19,279 [DEBUG] > User-Agent: Python/3.10 websockets/10.3
2025-06-20 02:49:20,344 [DEBUG] < HTTP/1.1 101 Switching Protocols
2025-06-20 02:49:20,345 [DEBUG] < Date: Fri, 20 Jun 2025 02:49:20 GMT
2025-06-20 02:49:20,346 [DEBUG] < Connection: upgrade
2025-06-20 02:49:20,347 [DEBUG] < Upgrade: websocket
2025-06-20 02:49:20,348 [DEBUG] < Sec-WebSocket-Accept: lqHTtoQr12nzZ+8KscFE18yyO2o=
2025-06-20 02:49:20,349 [DEBUG] < Sec-WebSocket-Extensions: permessage-deflate
2025-06-20 02:49:20,350 [DEBUG] < Strict-Transport-Security: max-age=31536000;
includeSubDomains; preload
2025-06-20 02:49:20,351 [DEBUG] < cf-cache-status: DYNAMIC
2025-06-20 02:49:20,352 [DEBUG] < Set-Cookie:
__cf_bm=C1z2FGv5ZL0d8CLEC3xBX5gElD4ZeWM99V5IN_P2ebU-1750387760-1.0.1.1-
DqQmMLqeR3qsxtJ08zzA3b58.mR6a8F28qUSmNEUVw8HYR6T7LFaFOi8ADGNGeY5GOI2gZJosurtKqHS3xlRdU1Egu
Woa5iYb6Lusy_LLbo; path=/; expires=Fri, 20-Jun-25 03:19:20 GMT; domain=.api.openai.com;
HttpOnly; Secure; SameSite=None
2025-06-20 02:49:20,353 [DEBUG] < X-Content-Type-Options: nosniff
2025-06-20 02:49:20,354 [DEBUG] < Set-Cookie:
_cfuvid=tsMNwObNWljvrmcLbTeX0xxz69OfMAM5oZuKdA95j8s-1750387760136-0.0.1.1-604800000;
path=/; domain=.api.openai.com; HttpOnly; Secure; SameSite=None
2025-06-20 02:49:20,355 [DEBUG] < Server: cloudflare
2025-06-20 02:49:20,356 [DEBUG] < CF-RAY: 9527f3c80f5bf1e0-KHH
2025-06-20 02:49:20,357 [DEBUG] < alt-svc: h3=":443"; ma=86400
2025-06-20 02:49:20,359 [DEBUG] = connection is OPEN
2025-06-20 02:49:20,362 [DEBUG] < TEXT
'{"type":"session.created","event_id":"event_BkM...cing":null,"tools":[]}}' [1210 bytes]
2025-06-20 02:49:20,366 [DEBUG] > TEXT '{"type": "session.update", "session":
{"input_a...ce_duration_ms": 500}}}' [214 bytes]

2025-06-20 02:49:20,370 [INFO] ==  Please speak into the microphone...  ==
```

5. 修订纪录

Date	Revision	Description
2025.07.04	1.00	初始发布。

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.