在 MA35D1 上實現即時語音對話的 ChatGPT

NuMicro® 64 位元系列微控制器範例程式碼介紹

文件資訊

應用簡述	在 Nuvoton OpenWrt 平台上，利用 OpenAI Realtime API 實現支援語音輸入與語音輸出的即時回應型 ChatGPT，提供自然順暢的語音對話體驗。
BSP 版本	MA35D1 Series OpenWrt 22.03
開發平台	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

本範例程式展示如何在 Nuvoton OpenWrt 嵌入式平台上，利用 Python 語言與 WebSocket 技術，結合 OpenAI 最新推出的 Realtime API，實現即時語音互動式對話系統。使用者透過麥克風說話後，裝置即時錄音並上傳音訊，OpenAI 模型則快速回覆語音內容，並由系統即時播放，使人機互動更加自然、快速、直覺。

這種低延遲互動能力，讓嵌入式平台也能具備建置智慧語音助手、語音客服、教育對話機器人等場景的潛力。即便在資源受限的平台上，透過雲端 AI 能力，仍可提供媲美手機等高階裝置的語音互動體驗。

此範例程式是基於 Nuvoton OpenWrt 22.03 的環境來進行開發，需先建立 Nuvoton OpenWrt 22.03 開發環境（https://github.com/OpenNuvoton/Nuvoton-OpenWrt-22.03），才能夠執行此範例程式。

雖然在這篇文章中，是以 NuMaker-IoT-MA35D16F90 V2.4 開發板來舉例。但是這篇文章的內容適用於所有支援 Nuvoton OpenWrt 22.03 且執行 Python 程式順暢的開發板，包含 MA35D0、MA35D1 和 MA35H0 等系列。

1.1 原理

OpenAI Realtime API 是 OpenAI 為即時語音互動場景所設計的新一代串流式介面，讓開發者能以語音輸入、語音輸出的方式與 GPT-4o 模型進行即時對話。它採用 WebSocket / WebRTC 雙向通訊機制，整合了語音辨識（ASR）、語意理解（LLM）與語音合成（TTS）於單一連線中，實現真正的低延遲語音互動體驗。

相較於傳統以 HTTP 為主的 Chat API，Realtime API 支援持續串流音訊，在使用者說話的同時即開始進行辨識與理解，不需等完整語句輸入完畢，能夠有效縮短回應時間至數百毫秒等級。

本範例程式核心原理為結合本地語音裝置（arecord / aplay），使用 Python 搭配 WebSocket 技術，串接 OpenAI Realtime API 與 OpenAI 伺服器進行溝通，實現雙向即時語音對談。整體流程如下：

1. 語音輸入處理：透過 Linux 的 arecord 工具進行錄音，以擷取語音資料。

2. 音訊串流上傳：錄音資料經 Base64 編碼後，即時送出至 OpenAI Realtime API（gpt-4o-realtime-preview）。
3. 語音辨識與理解：OpenAI 伺服器根據串流語音進行即時語音辨識，並根據辨識結果回覆語音回答與文字轉錄。
4. 語音回應處理：接收到的語音回應（Base64 編碼的語音片段）經解碼後送至 aplay 播放輸出。
5. 多輪對話支持：程式保留使用者提問與 GPT 回覆內容，以維持對話上下文，提升互動連貫性。

程式中利用 Python websockets 套件建立與 OpenAI 伺服器的雙向 WebSocket 連線，並採用 asyncio 架構設計「錄音傳送」、「回應接收」與「音訊播放」三個非同步工作程序，確保全流程的非阻塞處理與即時回應。

另外為了更穩定的語音偵測，範例程式採用伺服器端語音起止偵測（server_vad），當語音停止時自動中止錄音。這樣除了不需要按鍵操作來控制錄音的開始及結束，還可節省資源並提升互動效率。

相較於傳統透過文字傳送請求的 ChatGPT 互動模式，此範例程式使用語音為主的方式能更貼近自然語言溝通習慣，亦能應用於無螢幕或免觸控設備，讓嵌入式系統也能提供現代語音 AI 體驗。

1.2 環境設定

本範例程式以 Python 語言撰寫，執行過程中會使用多個 Python 模組與 Linux 工具。因此，在執行前必須先確認這些相依模組與工具均已安裝並可正常運作。由於 OpenWrt 系統本身可輕鬆支援這些必要工具，因此本說明將以 OpenWrt 作為開發與執行環境。

關於 MA35D1 平台的建置流程與映像檔產出方式，可參考 Nuvoton OpenWrt 22.03 Project User Manual。

- **OpenWrt 設定步驟**

在 OpenWrt 系統中，請依下列步驟於 menuconfig 中啟用本範例所需的套件：

1. 執行 make menuconfig 命令以開啟 OpenWrt 套件選單。可透過方向鍵選擇分類，並按空白鍵啟用套件。也可按「/」鍵並輸入套件名稱快速搜尋。

2. 本範例程式是以 Python 編寫，並透過 WebSocket 與 OpenAI Realtime API 建立連線，故需啟用下列 Python 相關套件：

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

3. 語音輸入與輸出部分使用 arecord 與 aplay 工具，需啟用 ALSA 音效支援相關套件：

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

4. 啟用所有必要套件後，請離開 menuconfig 並儲存設定。

5. 將 *chatgpt_audio.py* 檔案拷貝到 *target/linux/ma35d1/base-files/* 目錄，使其在系統建置完成後可直接位於 OpenWrt 的根目錄中。

6. 執行 make 命令進行系統建置，使用更新後的設定生成映像檔。

建置完成後，可透過 NuWriter 將映像檔燒錄至 NuMaker-IoT-MA35D16F90 開發板中，完成環境部署。

1.3 執行結果

本範例程式的 OpenWrt 映像檔執行編譯結果，如圖 1-1 所示。在編譯完成後，可以在 *bin/targets/ma35d1/iot/* 目錄下找到產生的映像檔。

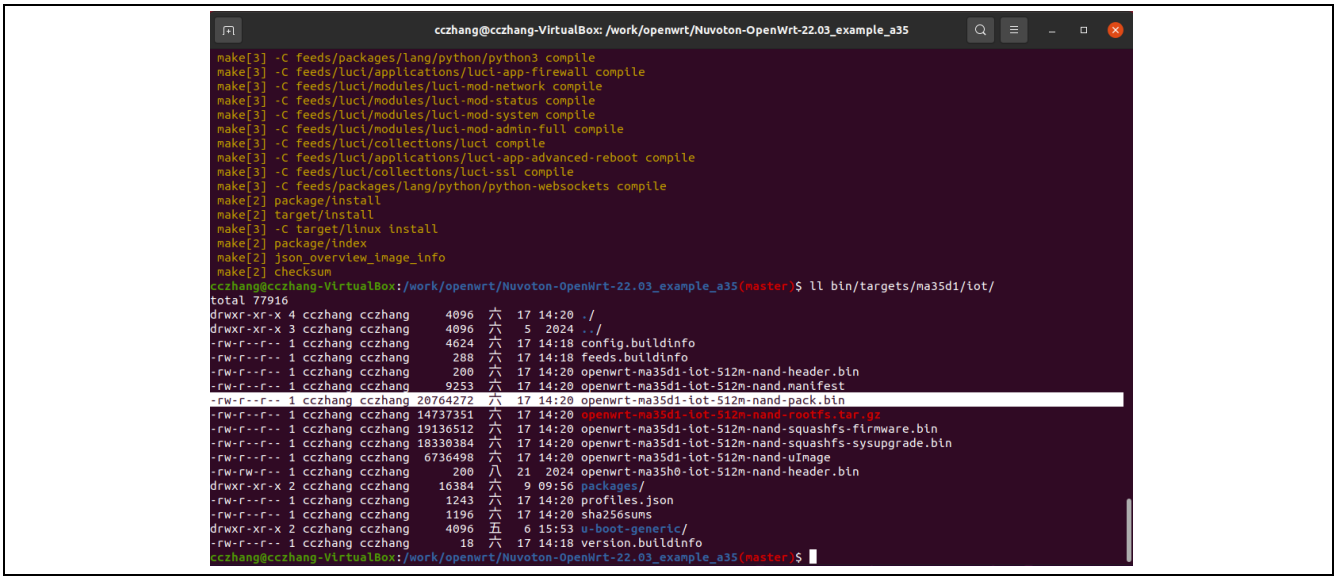

圖 1-1 OpenWrt 執行編譯結果

本範例程式的 OpenWrt 映像檔執行燒錄結果，如圖 1-2 所示。

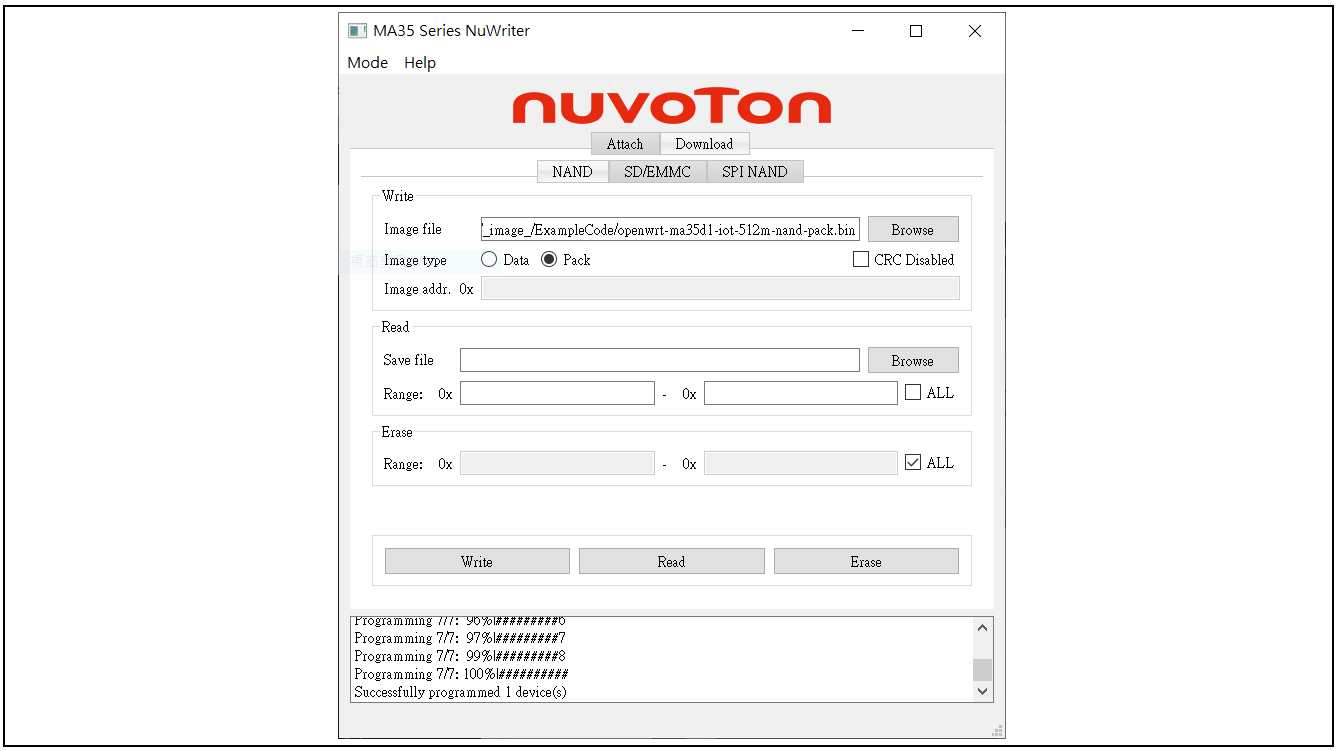

圖 1-2 使用 NuWriter 燒錄 OpenWrt 映像檔

本範例程式的執行流程與結果如下。當程式啟動後，畫面上會顯示「== Please speak into the microphone... ==」提示，使用者即可開始對著麥克風進行語音輸入。系統採用語音活動偵測（VAD）來判斷語音的開始與結束，依據語音與靜音的間隔進行判斷。因此若想連續說出多個句子，中間僅能停頓短暫時間，否則會被視為語音輸入已完成。

當畫面顯示「>> Speech started...」時，表示伺服器已偵測到語音輸入開始；而出現「<< Speech stopped...」時，則代表伺服器判定語音輸入已結束。接著，伺服器會回傳對應的語音與文字回應內容。文字通常會在短時間內全部顯示完畢，而語音則需實際依序播放每段音訊，播放過程無法加速，因此整體所需時間較長。當畫面顯示「Response done for <...>」時，代表此次提問的回應已生成完畢，但仍需等語音播放結束後，才能進行下一輪提問。

以下為實際執行時的終端機輸出訊息：

```
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?>
```

本範例程式在終端軟體中執行語音聊天功能時的輸出結果，這裡以 PuTTY 為例，如圖 1-3 所示。gpt-4o-realtime-preview 模型具備自動語言辨識能力，能夠支援多語言混合輸入與不同口音辨識。因此，使用者在與 ChatGPT 互動過程中，可以自由切換語言，例如中文、英文或日文，系統皆能正確辨識並給出相應語言的回應。

如圖所示，使用者可依序以不同語言提問，系統會即時顯示語音輸入、辨識、回應等過程，並根據輸入語言自動切換回應內容。這大幅提升了跨語言溝通的彈性與便利性，適合多語環境下的應用需求。

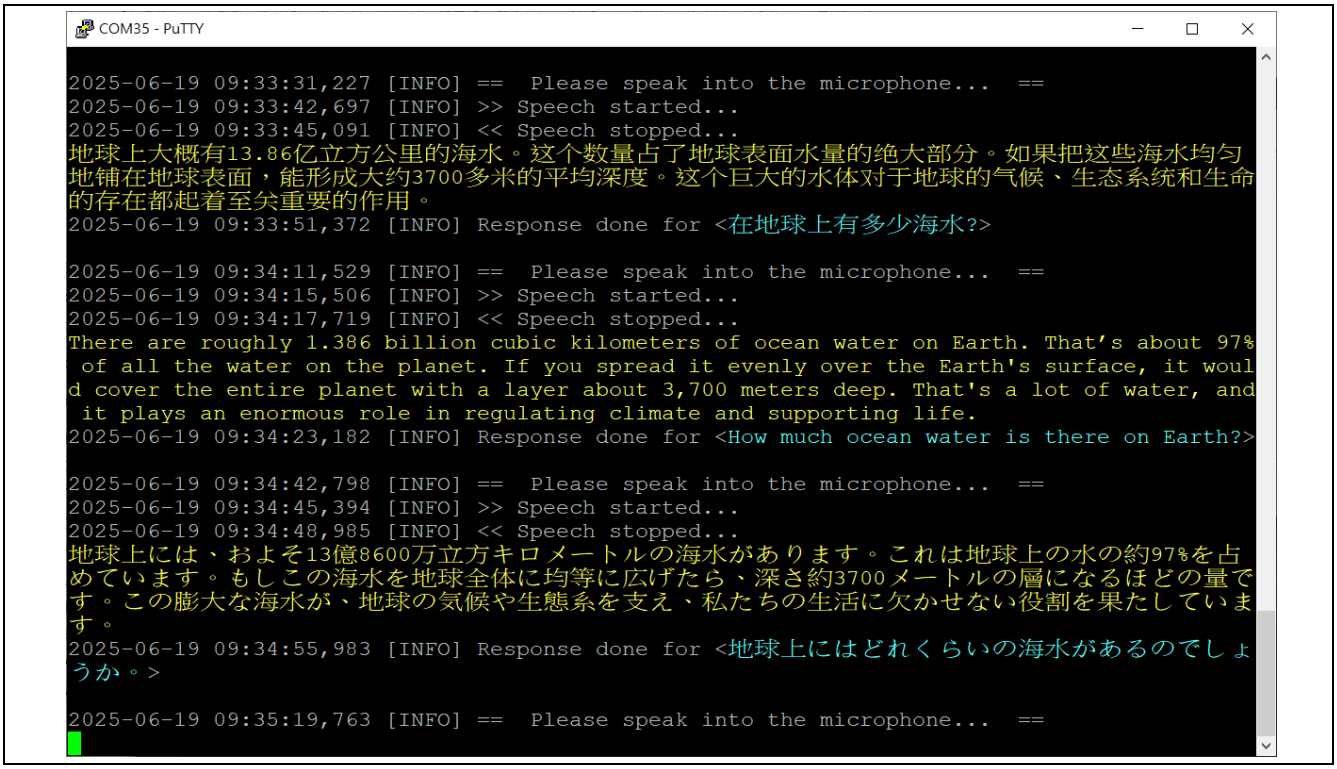

圖 1-3 ChatGPT 聊天功能範例 1

gpt-4o-realtime-preview 模型具備強大的上下文記憶能力，能夠支援多輪對話。這表示每次模型生成回應時，不僅僅考慮使用者剛剛輸入的內容，還會參考整個對話過程中的歷史紀錄，進行語意判斷與回應生成。使用者可以連續提出相關問題，系統會根據上下文脈絡給出更自然且連貫的回答。

如圖 1-4 所示，使用者可以多次與 ChatGPT 互動，針對先前的對話內容進行追問或延伸討論，gpt-4o-realtime-preview 模型皆能正確理解並延續對話，展現多輪互動的能力。

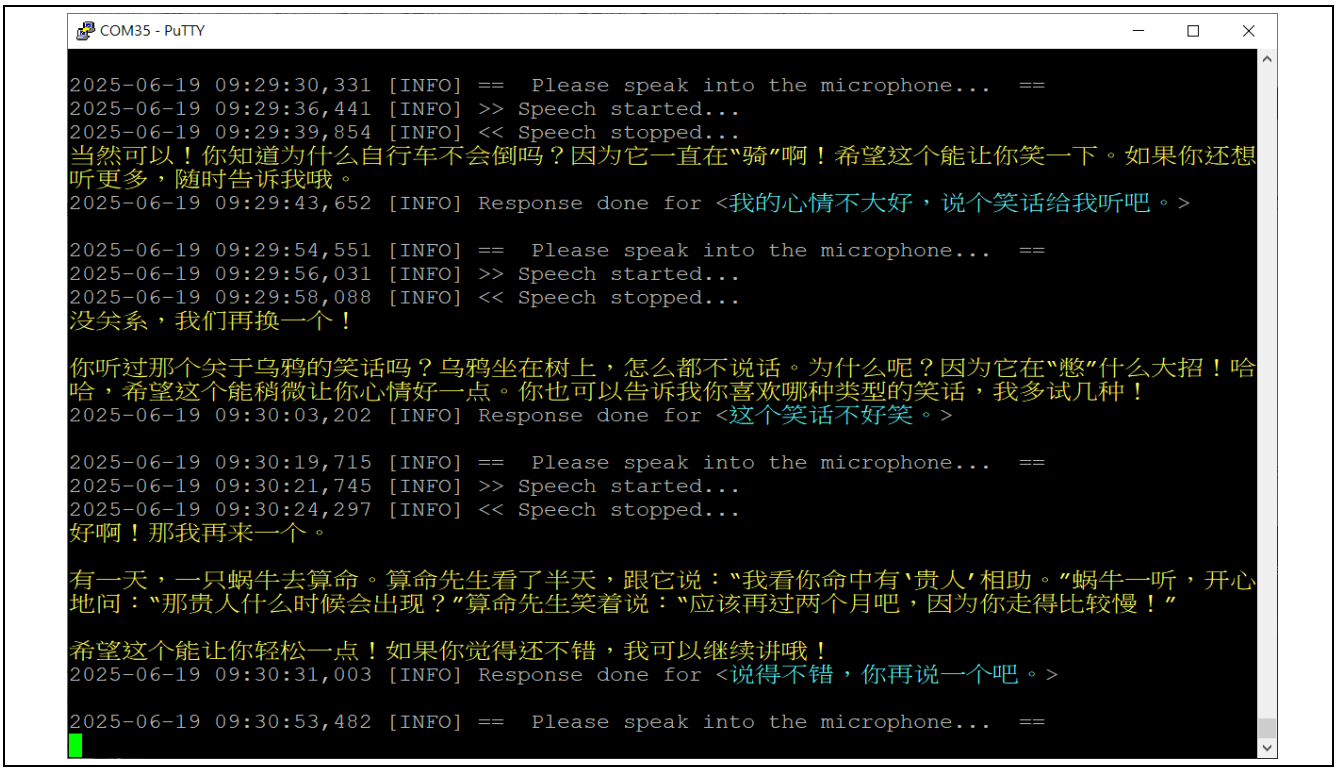

圖 1-4 ChatGPT 聊天功能範例 2

gpt-4o-realtime-preview 模型不僅能夠進行日常對話，也具備處理複雜條件與展現創意的能力。只要使用者給予明確的目標或指示，模型便能根據需求，生成結構完整且內容豐富的回應。這使得該系統能夠應用於故事創作、規劃建議、條列分析等多種場景。

如圖 1-5 所示，使用者可針對特定主題或條件提出請求，gpt-4o-realtime-preview 模型會根據指示自動產生詳盡且具創意的內容，展現強大的語言生成能力。

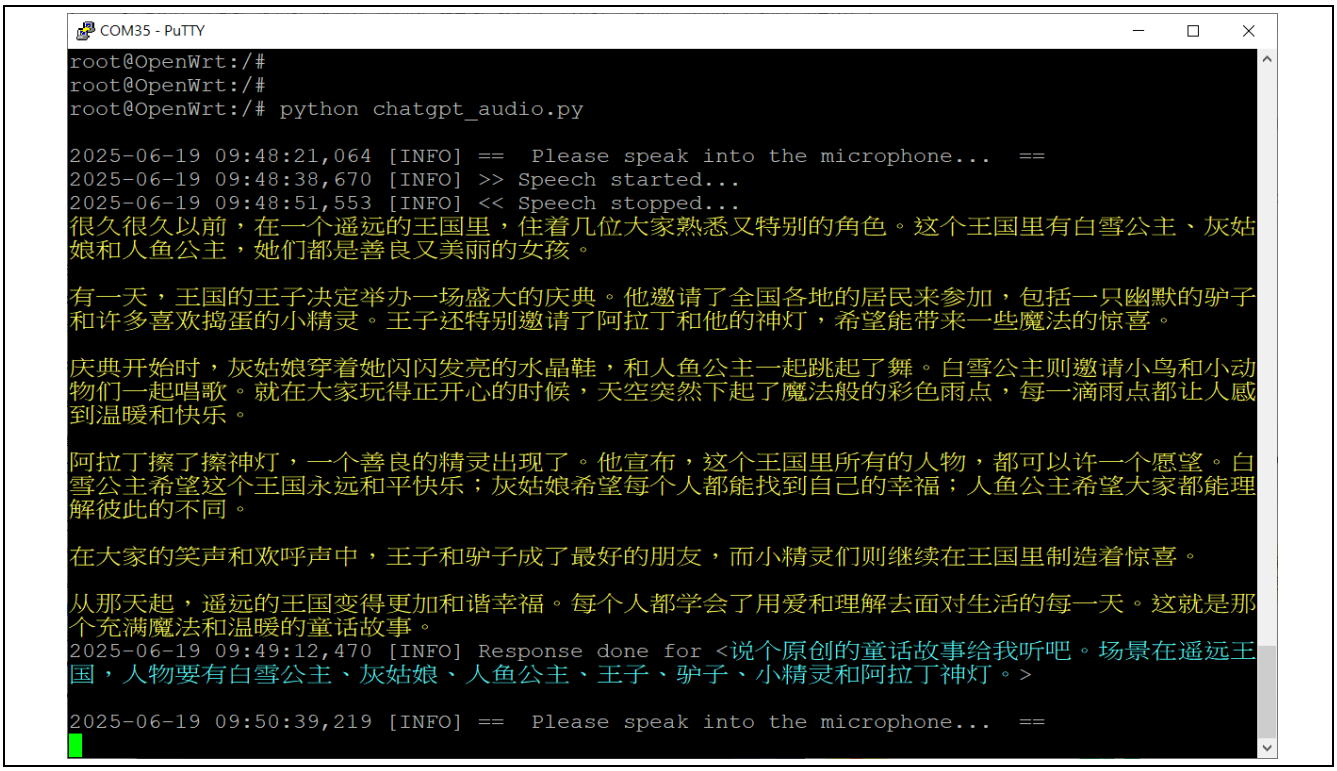

圖 1-5 ChatGPT 聊天功能範例 3

2. 程式碼介紹

在程式初始化階段，透過 logging.basicConfig() 設定日誌輸出格式與層級。預設層級為 INFO，可顯示基本執行資訊；若將層級設為 DEBUG，則可輸出更詳細的除錯訊息，包含非同步任務狀態、事件內容等，有助於開發與問題追蹤。此外，DEBUG 模式也會顯示其他模組（如 websockets、asyncio）的內部訊息，便於觀察整體系統行為。

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

AudioStreamer 類別為整個語音互動流程的核心，負責處理音訊串流、WebSocket 傳輸、語音播放與對話狀態管理。類別建立時會初始化數個屬性，其中 question 用於儲存使用者語音輸入最終辨識出的提問文字；full_transcript 則累加來自伺服器的語音辨識增量回應，以重建完整的語音輸入內容。ws 為 WebSocket 連線物件，負責與 OpenAI Realtime API 建立雙向通訊；_arecord_proc 用於控制錄音子程序；audio_play_queue 則為非同步佇列，負責將語音回應資料傳送至播放流程。

```
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
```

當 WebSocket 連線建立完成後，update_session() 方法會向 OpenAI Realtime API 傳送一段「session.update」訊息，用以設定語音轉文字模型與語音起止偵測（turn detection）的參數。此範例指定使用 gpt-4o-transcribe 模型進行語音辨識，並啟用伺服器端的語音活動偵測（Server VAD），使系統能自動判定使用者何時開始與結束說話，無需在裝置端額外處理啟停邏輯，提升語音互動的即時性與穩定性。

其中 turn detection 區塊的各個變數說明如下：

- type：設為 server_vad，啟用伺服器端語音起止判斷機制，由伺服器自動偵測語音輸入區段。

- threshold：語音偵測的靈敏度閾值，範圍介於 0.0 到 1.0。數值越低（如 0.3），越容易判定為開始說話，適合安靜環境；數值越高（如 0.7），可減少背景噪音誤觸，適合嘈雜環境。本範例設為 0.5，屬於中等靈敏度。
- prefix_padding_ms：當伺服器偵測到語音開始時，回補指定毫秒數的音訊作為語句開頭，防止語音截斷。本範例設為 800 毫秒，適合一般語速；語速較慢的使用者可增加至 1000 毫秒。
- silence_duration_ms：當偵測到使用者持續靜音超過此毫秒數後，即視為語音輸入結束。此範例設為 500 毫秒，以加快回應速度；縮短至 300 毫秒可進一步提升速度，但可能導致語音過早截斷。

```
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() 方法負責啟動錄音程式 arecord，並將錄製的語音資料即時串流傳送至 OpenAI Realtime API。此方法在執行時會啟動一個子程序，從標準輸出讀取錄音資料，每次讀取一小段音訊後，會先將資料轉換為 Base64 編碼，再透過 WebSocket 傳送至伺服器，事件類型為「input_audio_buffer.append」。該程序會持續執行，直到伺服器端的語音活動偵測（VAD）判定使用者已說完話，並由其他協同任務主動中止錄音子程序。整體流程確保使用者的語音輸入在說話過程中能穩定且即時地傳送至伺服器，為語音辨識與後續回應提供即時且連續的資料來源。

```
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() 方法負責播放模型回傳的語音回應資料，實作上會啟動 aplay 播放程式，並透過非同步佇列 audio_play_queue 持續接收音訊資料。該方法會等待佇列中出現新的語音片段，每取得一筆資料後便立即寫入 aplay 的輸入管線進行播放。播放過程會持續進行，直到佇列中收到 None 作為終止訊號，表示該回合語音資料的傳送已完成。此時程式會關閉輸入管線，並且等待 aplay 播放完最後一筆音訊資料後再結束程序，以確保語音內容完整播放而不被截斷。

```
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() 方法負責接收並處理來自 OpenAI Realtime API 的各類 WebSocket 訊息，包含語音偵測、語音辨識、語音回應與互動結束等事件。方法內透過非同步迴圈不斷監聽並解析伺服器回傳的訊息，並依據不同事件類型進行相對應的處理流程。

當接收到下列事件時，程式會做出以下動作：

- session.created：表示 WebSocket 工作階段已成功建立，系統可開始傳送語音資料。
- input_audio_buffer.speech_started：伺服器偵測到使用者開始說話，系統會於終端輸出提示，顯示語音錄製已啟動。
- input_audio_buffer.speech_stopped：伺服器偵測到使用者停止說話，此時會立即中止錄音子程序（arecord），釋放系統資源。
- response.audio_transcript.delta：模型持續回傳的語音辨識增量文字會被串接至 full_transcript，並即時顯示於畫面上。
- response.audio.delta：回傳模型語音回應資料的片段，系統會解碼並送入播放佇列，由播放任務處理輸出。
- response.done：表示該輪語音互動已結束，系統會發送終止訊號（None）至播放佇列，結束本次回應流程。
- conversation.item.input_audio_transcription.completed：回傳整段語音輸入的最終辨識結果，此文字會被追加至 question 作為本輪提問內容。

此外，若 WebSocket 連線被中斷（如伺服器關閉或網路中斷），系統會捕捉例外，自動結束此接收任務，避免整體應用程式錯誤中止。

整體來說，此方法為語音互動系統的接收核心，負責即時處理與驅動對話流程的各項關鍵事件，確保整體互動行為具備完整性、即時性與同步性。

```
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() 方法為整個程式的主要進入點，負責建立與管理 WebSocket 連線、初始化非同步工作流程（錄音、接收、播放），並處理回合控制與錯誤復原。程式會進入一個永久迴圈，嘗試與 OpenAI Realtime API 建立 WebSocket 連線，一旦連線成功，便呼叫 update_session() 方法傳送初始設定，並準備進行一次語音互動。

每一輪互動開始時，會重設輸出佇列 audio_play_queue，並啟動三個非同步任務：audio_send_worker() 負責送出語音資料、ws_receive_worker() 接收伺服器回應事件、audio_play_worker() 播放語音回應。這些任務會並行執行，直到接收到伺服器的結束訊號為止。

互動結束後，程式會等待各任務完成，並傳送終止訊號以結束播放流程。如果連線中途中斷，將進入例外處理區塊，稍作延遲後自動重新連線。最後，透過 finally 區塊完成錄音與播放程序的清除、任務取消、佇列釋放與連線關閉，確保系統穩定且能重新啟動下一輪互動。

```
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() 方法為程式的啟動入口，會先確認環境變數 OPENAI_KEY 是否存在，若未設定則顯示錯誤並結束程式。若檢查通過，則建立 AudioStreamer 物件並執行 run_loop() 方法，啟動語音互動流程。整體流程包在 try-except 區塊中，若發生例外錯誤，將輸出錯誤訊息與堆疊追蹤，方便後續除錯。

```
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. 軟體與硬體需求

3.1 軟體需求

- BSP 版本
  - MA35D1 Series OpenWrt 22.03

3.2 硬體需求

- 電路元件
  - NuMaker-IoT-MA35D16F90 V2.4

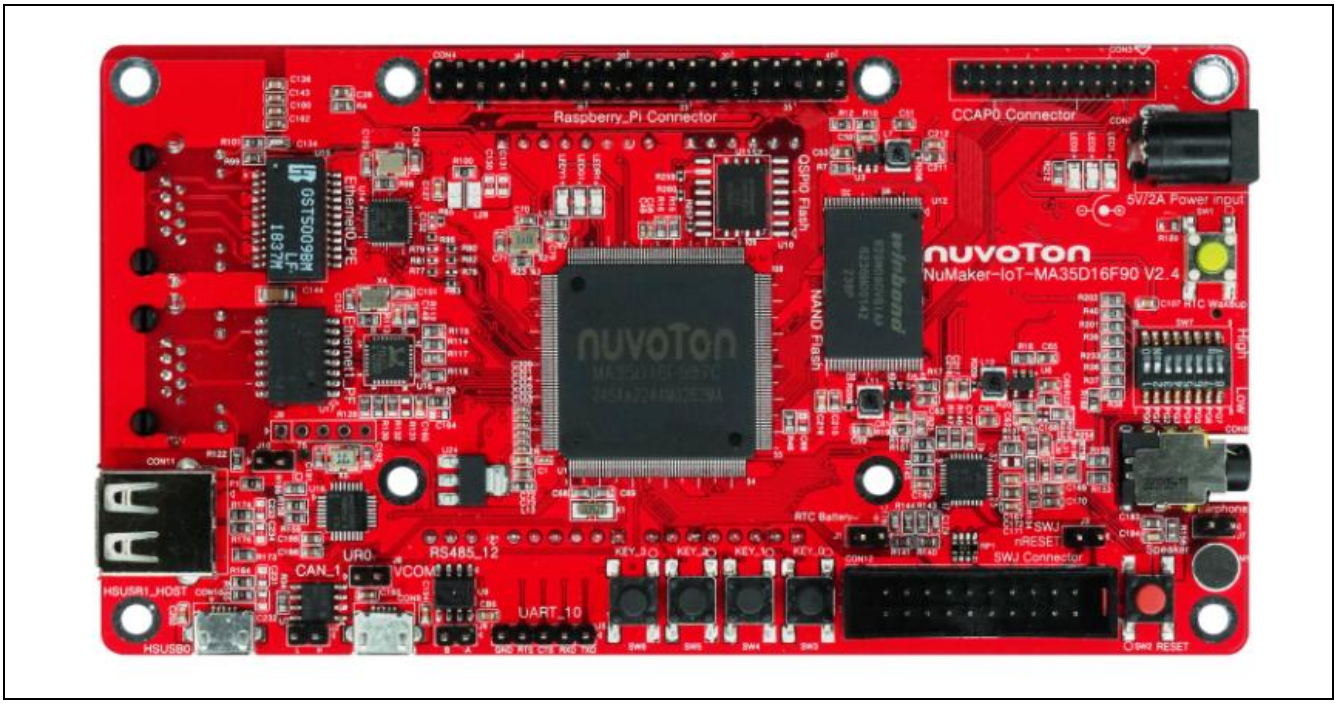

圖 3-1 NuMaker-IoT-MA35D16F90 V2.4

- 語音設備
  - TRRS 接頭耳機麥克風

4. 範例程式執行

1. 使用終端機軟體連接 NuMaker-IoT-MA35D16F90 開發板的 VCOM。

2. 將具備 TRRS 接頭的耳機麥克風插入開發板。

3. 系統啟動後，請確認裝置已成功連上網路，且系統時間正確，這是正常執行 ChatGPT 範例程式的前提。可使用 ping 與 date 指令進行確認：

```
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. 執行前可先調整耳機的音量至適當大小：

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

5. 設定 OpenAI API 金鑰環境變數（請將 your_key_here 替換為實際金鑰）：

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

6. 執行範例程式，即會進入語音交談模式。當畫面出現提示訊息時，即可開始語音輸入，流程如前述章節所示：

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

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

7. 若需要除錯，可將程式中的 logging 等級從 INFO 改為 DEBUG。啟用後，執行時將顯示更詳細的偵錯訊息，方便觀察連線與事件流程：

```
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. 修訂紀錄

Date	Revision	Description
2025.07.04	1.00	初始發布。

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.