Implementing Real-time Voice-based ChatGPT on MA35D1

Example Code Introduction for 64-bit NuMicro® Family

Document Information

Application	On the Nuvoton OpenWrt platform, this project leverages the OpenAI Realtime API to implement a real-time, voice-interactive ChatGPT that supports both voice input and voice output, delivering a natural and smooth conversational experience.
BSP Version	MA35D1 Series OpenWrt 22.03
Hardware	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. Overview

This example code demonstrates how to build a real-time, voice-interactive conversation system on the Nuvoton OpenWrt embedded platform using Python and WebSocket technologies, integrated with OpenAI's latest Realtime API. After a user speaks into a microphone, the device records and uploads the audio in real time. The OpenAI model then quickly generates a voice response, which is immediately played back by the system, making human-computer interaction more natural, fast, and intuitive.

This low-latency interaction capability enables embedded platforms to support applications such as intelligent voice assistants, voice-based customer service, and educational chatbots. Even on resource-constrained devices, leveraging cloud-based AI allows for a voice interaction experience comparable to that of high-end devices like smartphones.

This example code is developed based on the Nuvoton OpenWrt 22.03 environment. You must first set up the Nuvoton OpenWrt 22.03 development environment (https://github.com/OpenNuvoton/Nuvoton-OpenWrt-22.03) before running this example.

While this document uses the NuMaker-IoT-MA35D16F90 evaluation board as an example, the content applies to any evaluation board that supports Nuvoton OpenWrt 22.03 and can run Python smoothly, including the MA35D0, MA35D1, and MA35H0 series.

1.1 Principle

The OpenAI Realtime API is a next-generation streaming interface designed by OpenAI specifically for real-time voice interaction scenarios. It enables developers to engage in real-time conversations with the GPT-4o model using both voice input and voice output. Utilizing a bidirectional communication mechanism based on WebSocket/WebRTC, it integrates automatic speech recognition (ASR), large language model (LLM) understanding, and text-to-speech (TTS) synthesis within a single connection, delivering a truly low-latency voice interaction experience.

Compared to traditional HTTP-based Chat APIs, the Realtime API supports continuous audio streaming, allowing recognition and understanding to begin as soon as the user starts speaking—without waiting for the entire utterance to be completed. This significantly reduces response times to the scale of a few hundred milliseconds.

The core principle of this example code is to combine local audio devices (arecord/aplay) with Python and WebSocket technology to connect to the OpenAI Realtime API and communicate with the OpenAI server, enabling real-time, bidirectional voice conversations. The overall workflow is as follows:

1. Voice Input Processing: Audio is recorded using the Linux arecord tool to capture voice data.
2. Audio Streaming Upload: The recorded audio is encoded in Base64 and streamed in real time to the OpenAI Realtime API (gpt-4o-realtime-preview).

3. Speech Recognition and Understanding: The OpenAI server performs real-time speech recognition on the streamed audio and generates both voice and text responses based on the recognized content.
4. Voice Response Handling: The received voice response (Base64-encoded audio segments) is decoded and played back using aplay.
5. Multi-turn Conversation Support: The program retains both user questions and GPT responses to maintain conversational context and enhance interaction continuity.

The code uses the Python websockets package to establish a bidirectional WebSocket connection with the OpenAI server and leverages the asyncio framework to implement three asynchronous tasks: audio streaming, response receiving, and audio playback. This ensures non-blocking processing and real-time responsiveness throughout the workflow.

For more stable voice detection, the example code adopts server-side voice activity detection (server_vad), which automatically stops recording when speech ends. This eliminates the need for manual button controls to start or stop recording, saving resources and improving interaction efficiency.

Compared to traditional text-based ChatGPT interaction modes, this example code's voice-centric approach is more aligned with natural language communication habits and can be applied to screenless or touch-free devices, enabling embedded systems to deliver a modern voice AI experience.

1.2 Environment Setup

This example code is written in Python and utilizes several Python modules and Linux tools during execution. Therefore, before running the program, you must ensure that all required dependencies and tools are properly installed and functioning. Since the OpenWrt system natively supports these essential tools, this guide uses OpenWrt as the development and execution environment.

For instructions on building the MA35D1 platform and generating the firmware image, please refer to the Nuvoton OpenWrt 22.03 Project User Manual.

- **OpenWrt Setup Steps**

In the OpenWrt system, please follow the steps below to enable the packages required for this example code in menuconfig:

1. Run the make menuconfig command to open the OpenWrt package selection menu. You can use the arrow keys to navigate categories and press the spacebar to enable packages. You can also press "/" and enter the package name to search quickly.
2. Since this example code is written in Python and uses WebSocket to connect to the OpenAI Realtime API, you need to enable the following Python-related packages:
    - Languages ---> Python ---> <*> python3..... Python 3.10 programming language

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

3. For voice input and output, the arecord and aplay tools are used, so you need to enable the ALSA sound support packages:

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

4. After enabling all necessary packages, exit menuconfig and save your configuration.

5. Copy the *chatgpt_audio.py* file to the *target/linux/ma35d1/base-files/* directory so that it will be placed in the root directory of OpenWrt after the system build is complete.

6. Run the "make" command to build the system and generate the firmware image using the updated configuration.

Once the build is complete, you can use NuWriter to program the firmware image to the NuMaker-IoT-MA35D16F90 evaluation board and complete the environment setup.

1.3 Demo Result

The compilation result of the OpenWrt firmware image for this example code is shown in Figure 1-1. After the build process is complete, you can find the generated firmware image in the *bin/targets/ma35d1/iot/* directory.

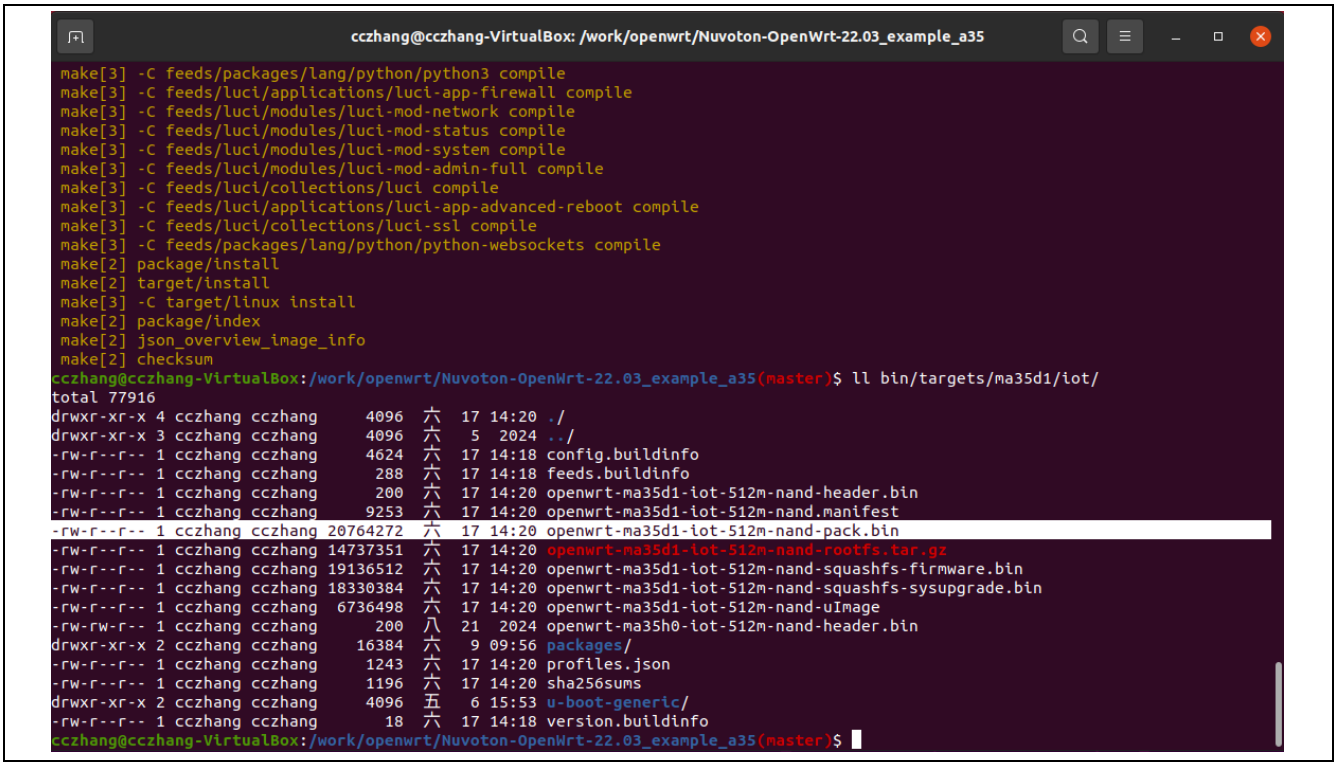

Figure 1-1 OpenWrt Build Results

The programming result of the OpenWrt firmware image for this example code is shown in Figure 1-2.

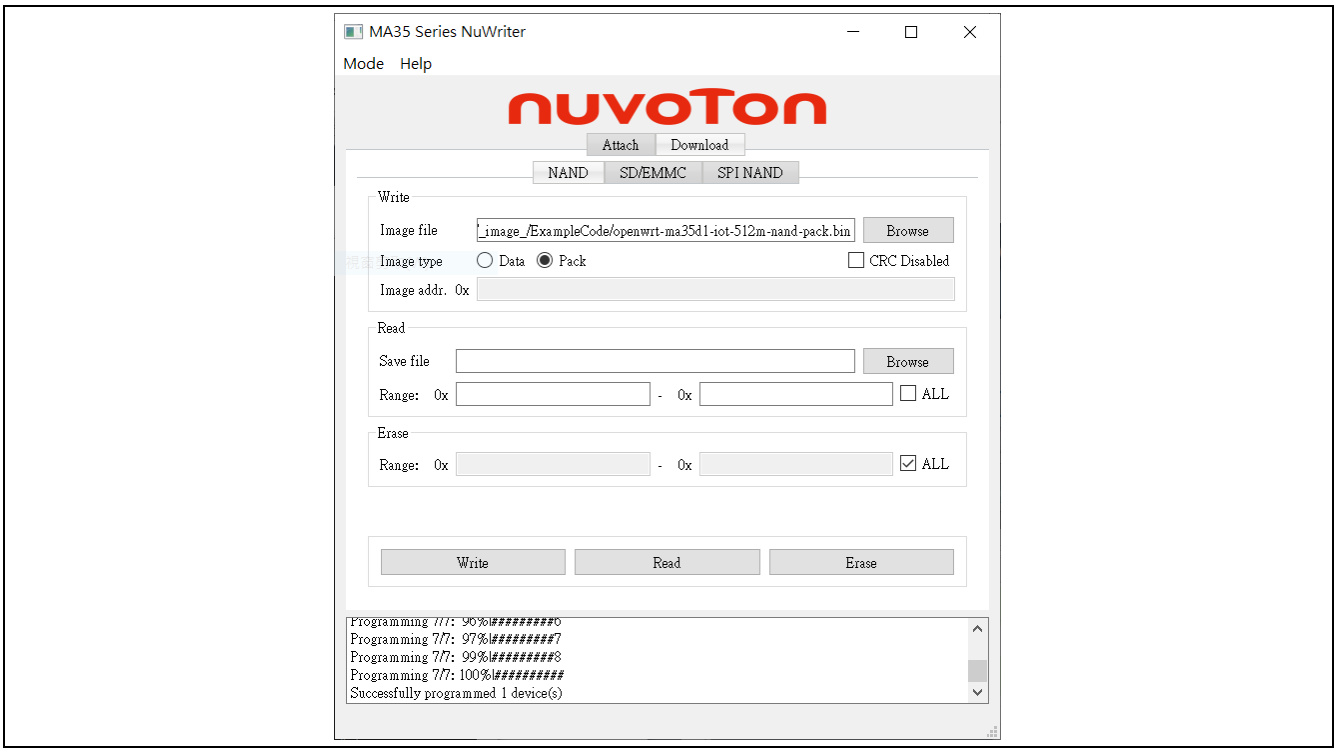

Figure 1-2 Programming the OpenWrt Firmware Image Using NuWriter

The execution flow and results of this example code are as follows. After the program starts, the screen will display the prompt "== Please speak into the microphone... ==", indicating that the user can begin speaking into the microphone for voice input. The system uses Voice Activity Detection (VAD) to determine the start and end of speech, based on the intervals between speech and silence. Therefore, if you want to say multiple sentences in a row, you can only pause briefly between them; otherwise, the system will consider the voice input as complete.

When the screen shows ">> Speech started...", it means the server has detected the start of voice input. When "<< Speech stopped..." appears, it indicates that the server has determined the end of the voice input. The server will then return the corresponding voice and text responses. The text response is usually displayed in full within a short time, while the voice response is played back sequentially and cannot be sped up, so the overall process takes longer. When the screen displays "Response done for <...>", it means the response to the current question has been generated, but you must wait for the voice playback to finish before starting the next round of input.

Below is an example of the actual terminal output during execution:

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

2025-06-26 10:38:43,304 [INFO] ==  Please speak into the microphone...  ==
2025-06-26 10:38:47,968 [INFO] >> Speech started...
2025-06-26 10:38:49,530 [INFO] << Speech stopped...
The total volume of ocean water on Earth is roughly 1.3 billion cubic kilometers. That’s
an immense amount—about 96.5% of all water on Earth is found in the oceans!
2025-06-26 10:38:57,515 [INFO] Response done for <How much ocean water is there on Earth?>
```

The output of the voice chat function executed by this example code in a terminal application (using PuTTY as an example) is shown in Figure 1-3. The gpt-4o-realtime-preview model features automatic language recognition, supporting mixed-language input and different accent detection. This allows users to freely switch between languages—such as English, Chinese, or Spanish—during interactions with ChatGPT, and the system will accurately recognize and respond in the corresponding language.

As shown in Figure 1-3, users can ask questions in different languages one after another. The system will display the processes of voice input, recognition, and response in real time, and automatically switch the response language according to the input. This greatly enhances the flexibility and convenience of cross-language communication, making it suitable for multilingual application scenarios.

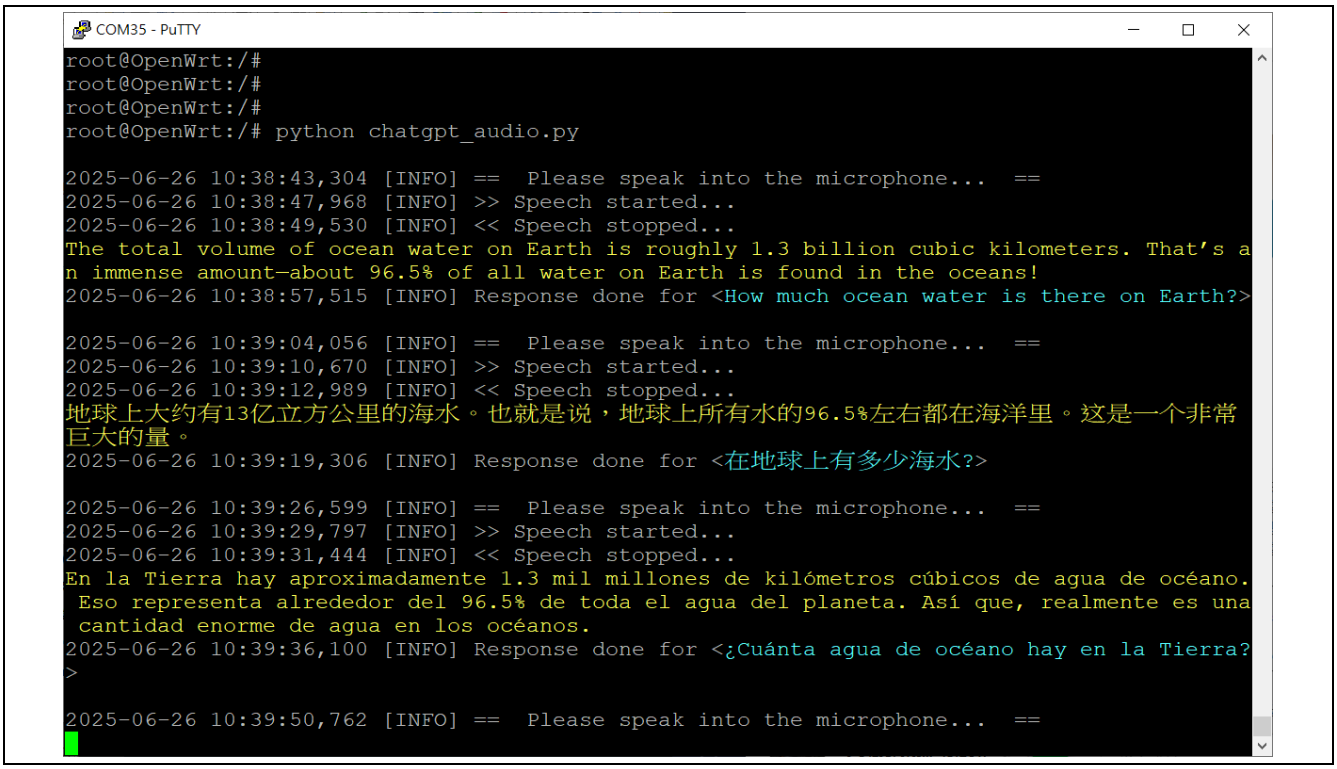

Figure 1-3 ChatGPT Chat Functionality Example 1

The gpt-4o-realtime-preview model features powerful contextual memory, enabling support for multi-turn conversations. This means that each time the model generates a response, it not only considers the most recent user input but also references the entire conversation history to perform semantic analysis and generate coherent replies. Users can ask follow-up or related questions consecutively, and the system will provide more natural and contextually connected answers.

As shown in Figure 1-4, users can interact with ChatGPT multiple times, asking follow-up questions or continuing discussions based on previous conversation content. The gpt-4o-realtime-preview model can accurately understand and maintain the flow of the conversation, demonstrating robust multi-turn interaction capabilities.

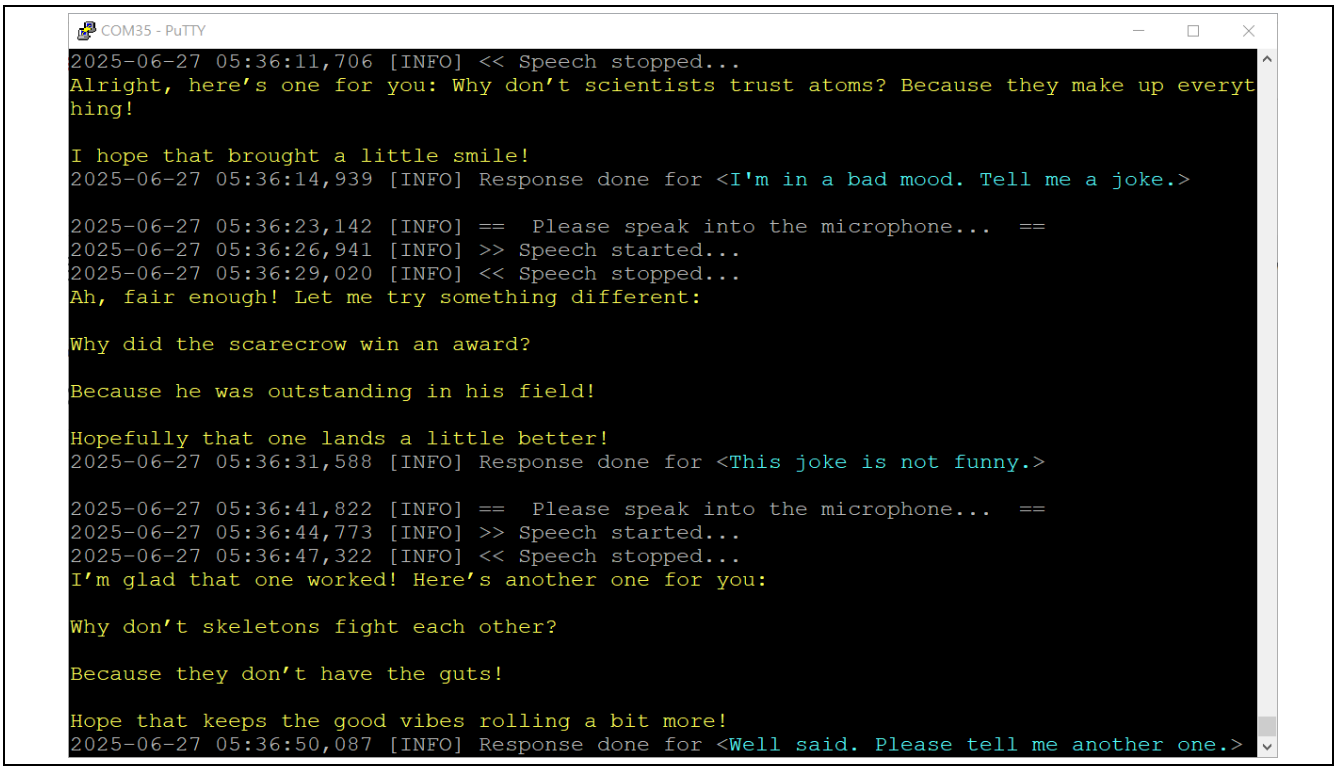

Figure 1-4 ChatGPT Chat Functionality Example 2

The gpt-4o-realtime-preview model is not only capable of handling everyday conversations, but also excels at processing complex requirements and demonstrating creativity. As long as users provide clear goals or instructions, the model can generate well-structured and content-rich responses tailored to specific needs. This enables the system to be applied in a variety of scenarios, such as story creation, planning suggestions, and analytical summaries.

As shown in Figure 1-5, users can make requests based on specific topics or conditions, and the gpt-4o-realtime-preview model will automatically generate detailed and creative content according to the instructions, showcasing its powerful language generation capabilities.

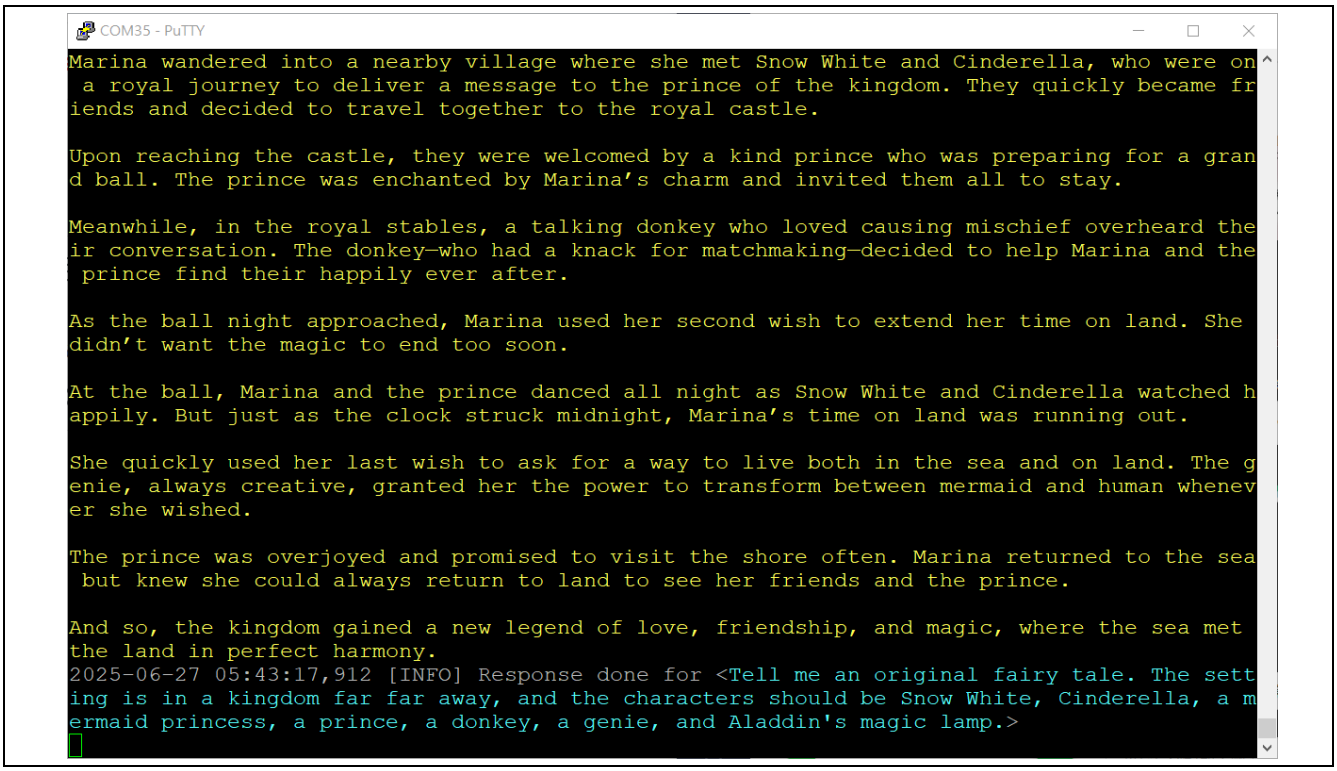

Figure 1-5 ChatGPT Chat Functionality Example 3

2. Code Description

During program initialization, logging.basicConfig() is used to configure the log output format and level. The default level is set to INFO, which displays basic execution information. If the level is set to DEBUG, more detailed debugging messages will be output, including the status of asynchronous tasks and event details, which is helpful for development and troubleshooting. Additionally, DEBUG mode will also display internal messages from other modules (such as websockets and asyncio), making it easier to observe the overall system behavior.

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

The AudioStreamer class serves as the core of the entire voice interaction process, handling audio streaming, WebSocket communication, voice playback, and conversation state management. When the class is instantiated, several attributes are initialized: question is used to store the final recognized user query from voice input, while full_transcript accumulates incremental speech recognition responses from the server to reconstruct the complete voice input. ws is the WebSocket connection object responsible for establishing bidirectional communication with the OpenAI Realtime API; arecord_proc is used to control the audio recording subprocess; and audio_play_queue is an asynchronous queue responsible for passing voice response data to the playback process.

```
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
```

After the WebSocket connection is established, the update_session() method sends a "session.update" message to the OpenAI Realtime API to configure the speech-to-text model and turn detection parameters. In this example, the gpt-4o-transcribe model is specified for speech recognition, and server-side Voice Activity Detection (Server VAD) is enabled. This allows the system to automatically determine when the user starts and stops speaking, eliminating the need for additional start/stop logic on the device side and improving the immediacy and stability of voice interactions.

The variables in the turn detection section are explained as follows:

- type: Set to server_vad to enable server-side speech boundary detection, allowing the server to automatically detect segments of voice input.
- threshold: The sensitivity threshold for voice detection, ranging from 0.0 to 1.0. Lower values (e.g., 0.3) make it easier to detect the start of speech, suitable for quiet environments; higher values (e.g., 0.7) help reduce false triggers from background

noise, suitable for noisy environments. In this example, it is set to 0.5, which is a medium sensitivity.

- prefix_padding_ms: When the server detects the start of speech, it prepends the specified number of milliseconds of audio to the beginning of the utterance to prevent speech from being cut off. This example uses 800 milliseconds, which is suitable for normal speaking speed; users who speak more slowly can increase this to 1000 milliseconds.
- silence_duration_ms: When the user remains silent for longer than this number of milliseconds, the system considers the voice input to be finished. In this example, it is set to 500 milliseconds to speed up responses; reducing it to 300 milliseconds can further increase speed but may cause speech to be cut off prematurely.

```
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)
                }
            }
        }))
```

The audio_send_worker() method is responsible for launching the arecord recording program and streaming the recorded audio data in real time to the OpenAI Realtime API. When this method is executed, it starts a subprocess that reads audio data from standard output. Each chunk of audio is first encoded in Base64 and then sent to the server via WebSocket with the event type "input_audio_buffer.append". This process continues until the server-side Voice Activity Detection (VAD) determines that the user has finished speaking, at which point another asynchronous task will actively terminate the recording subprocess. This workflow ensures that the user's voice input is transmitted to the server in a stable and real-time manner throughout the speaking process, providing a continuous and immediate data source for speech recognition and subsequent responses.

```
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,
        }))
```

The audio_play_worker() method is responsible for playing back the voice response data returned by the model. In implementation, it launches the aplay playback program and continuously receives audio data through the asynchronous queue audio_play_queue. This method waits for new audio segments to appear in the queue, and upon receiving each piece of data, it immediately writes it to the input stream of aplay for playback. The playback process continues until the queue receives None as a termination signal, indicating that all audio data for the current round has been sent. At this point, the program closes the input stream and waits for aplay to finish playing the last audio segment before ending the process, ensuring that the entire voice response is played back completely without being cut off.

```
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
```

The ws_receive_worker() method is responsible for receiving and processing various WebSocket messages from the OpenAI Realtime API, including events such as speech detection, speech recognition, voice response, and interaction termination. Within this method, an asynchronous loop continuously listens for and parses messages returned by the server, handling each event type accordingly.

When the following events are received, the program performs the corresponding actions:

- session.created: Indicates that the WebSocket session has been successfully established, and the system can begin sending audio data.
- input_audio_buffer.speech_started: The server detects that the user has started speaking; the system outputs a prompt in the terminal indicating that voice recording has started.
- input_audio_buffer.speech_stopped: The server detects that the user has stopped speaking; at this point, the recording subprocess (arecord) is immediately terminated to free system resources.
- response.audio_transcript.delta: Incremental speech recognition text returned by the model is appended to full_transcript and displayed on the screen in real time.
- response.audio.delta: Segments of the model's voice response data are returned, decoded by the system, and placed into the playback queue for the playback task to handle.
- response.done: Indicates that the current round of voice interaction has ended; the system sends a termination signal (None) to the playback queue to end the current response process.
- conversation.item.input_audio_transcription.completed: Returns the final recognition result of the entire voice input, which is appended to question as the content of the current query.

Additionally, if the WebSocket connection is interrupted (such as the server closing or a network disconnection), the system will catch the exception and automatically terminate this receiving task to prevent the entire application from crashing.

Overall, this method serves as the core receiver for the voice interaction system, responsible for real-time processing and driving key events in the conversation flow, ensuring the interaction is complete, immediate, and synchronized.

```
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")
```

The run_loop() method serves as the main entry point of the entire program. It is responsible for establishing and managing the WebSocket connection, initializing asynchronous workflows (recording, receiving, and playback), and handling session control and error recovery. The program enters a perpetual loop, attempting to establish a WebSocket connection with the OpenAI Realtime API. Once the connection is successful, it calls the update_session() method to send the initial configuration and prepares for a round of voice interaction.

At the start of each interaction round, the output queue audio_play_queue is reset, and three asynchronous tasks are launched: audio_send_worker() for sending audio data, ws_receive_worker() for receiving server response events, and audio_play_worker() for playing back voice responses. These tasks run concurrently until a termination signal is received from the server.

After the interaction ends, the program waits for all tasks to complete and sends a termination signal to end the playback process. If the connection is interrupted, the program enters an exception handling block, waits briefly, and then automatically attempts to reconnect. Finally, the finally block ensures cleanup of the recording and playback processes, task cancellation, queue release, and connection closure, maintaining system stability and enabling the next round of interaction to start smoothly.

```
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()
```

The main() method serves as the program's entry point. It first checks whether the environment variable OPENAI_KEY exists; if it is not set, an error message is displayed and the program exits. If the check passes, an AudioStreamer object is created and the run_loop() method is executed to start the voice interaction process. The entire flow is wrapped in a try-except block, so if an exception occurs, an error message and stack trace will be output to facilitate subsequent debugging.

```
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. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - MA35D1 Series OpenWrt 22.03

3.2 Hardware Requirements

- Circuit components
  - NuMaker-IoT-MA35D16F90 V2.4

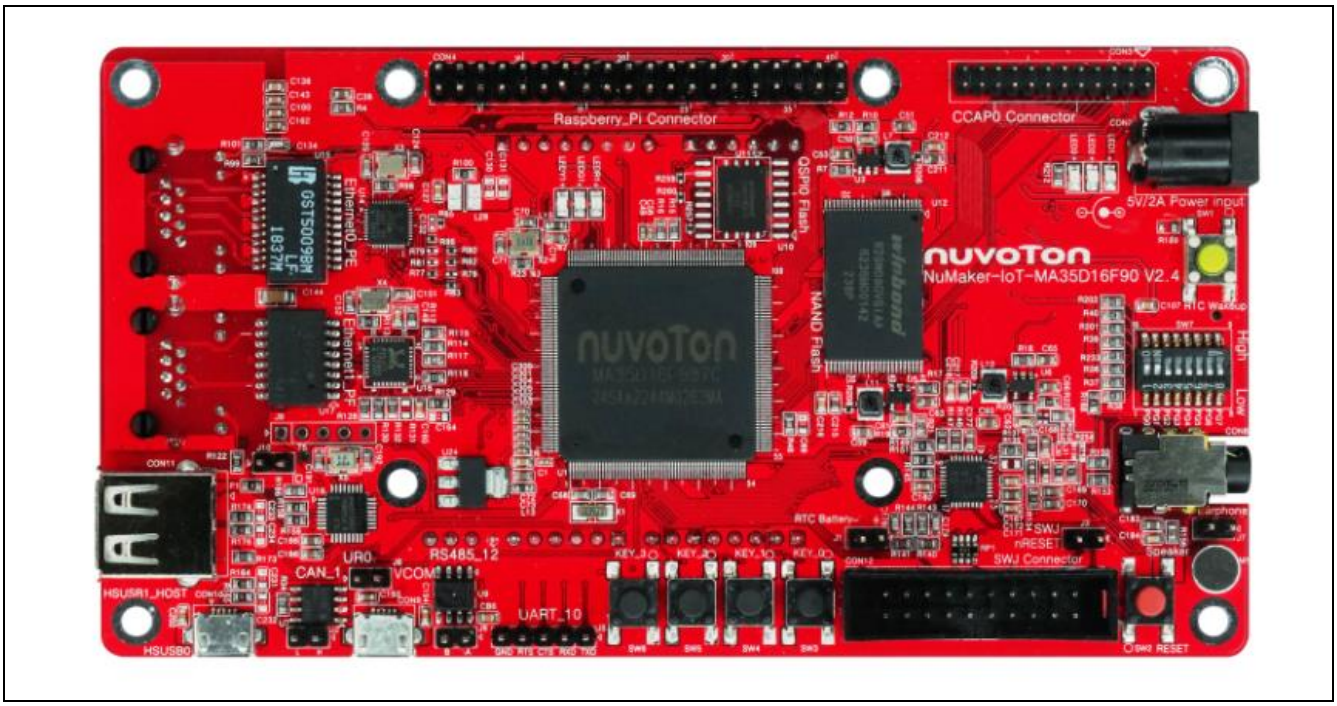

Figure 3-1 NuMaker-IoT-MA35D16F90 V2.4

- Audio device
  - Headset microphone with TRRS connector

4. Example Code Execution

1. Use terminal software to connect to the VCOM port of the NuMaker-IoT-MA35D16F90 evaluation board.
2. Plug a headset microphone with a TRRS connector into the evaluation board.
3. After the system starts, make sure the device is successfully connected to the network and the system time is correct. This is a prerequisite for running the ChatGPT example program properly. You can use the ping and date commands to verify:

```
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. Before running the program, you can adjust the headphone volume to an appropriate level:

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

5. Set the OpenAI API key environment variable (replace your_key_here with your actual key):

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

6. Run the example program to enter voice conversation mode. When the prompt appears on the screen, you can start voice input, as described in the previous sections:

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

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

7. If debugging is needed, you can change the logging level in the program from INFO to DEBUG. After enabling, more detailed debugging messages will be displayed during execution, making it easier to observe the connection and event flow:

```
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. Revision History

Date	Revision	Description
2025.07.04	1.00	Initial version.

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.