# Recording Audio with the Windows Waveform Audio API

This tutorial explains how to make audio recordings with a computer sound card using the Windows **Waveform Audio** API. This set of functions has been "deprecated", meaning that Microsoft discourages programmers from using this much older interface in favor of newer ones like **WASAPI** which are designed for newer "object-oriented" languages like C++. However, the older API, while a bit more primitive and low-level, does everything we need it to do to capture an audio stream, and it's easier to program. This tutorial will show how to use it in x86 assembly language. (The same method can be used with other languages.)

So what all is involved with making an audio recording? Before starting the actual recording process we need to find a **Wave device** that we can open: the "device" we'll be using here is a software device driver, which obviously needs to be talking to an actual hardware device, the sound card. This assumes that all this—the soundcard and its associated drivers—have been properly installed. For modern desktop computers, this will be the case most of the time.

For this tutorial we'll be using the line input to the sound card. Most sound cards also have a microphone input that can be used, and the microphone input could be used here by adapting the code. The first part of the tutorial will show how to make sure we open the right "device".

Here's the overall sequence for recording:

- **1.** Find the right device (in our case, the Wave "line-in" input device).
- **2.** Set up that device with the settings we want for the recording—sampling frequency, number of channels, bits per sample, etc.—and open the device with those settings.
- **3.** Set up the buffers we'll be using to record the audio data, and tell the Waveform Audio API about those buffers.
- **4.** Start the recording. While we're recording we need to manage the buffers, emptying them as they're filled by the device driver and making sure we always have new buffers lined up. This is covered below.
- **5.** End the recording and do something with the audio data: write it to disk, analyze it, whatever.

It's that middle part, managing the buffers, that's a little bit tricky, so most of this tutorial is focused on that aspect of recording.

## Finding and Opening The Device

There are two types of Waveform Audio devices, input and output. Since we're recording, we need an input device. (If we were playing an audio file we'd use an output device instead.)

A computer might have one or more input sound devices installed, so how do we find the one we need? In this example we're using the "line in" device, and the way we find it is by looking for that exact string in the device driver name. It turns out that that string is the very first part of the device driver's name. If I do a scan of available sound devices on my computer, this is the only input device I see (I wrote a little program to show this):

```
1 input Wave device(s) found:
     Driver: "Line In (High Definition Audio "
```

So before actually opening our device we need to do two things:

- Call **`waveInGetNumDevs()`** to find out how many input devices total are installed
- For each device in a loop, call **`waveInGetDevCaps()`** to see if it's the one we want

To find our line-in device we do a simple string comparison (case-insensitive) of the first part of the device-driver name with the string "**`line in`**", using the function **`strnicmp()`**. If it matches, that's our guy. (If you want to use the microphone input, use the string "**`microphone`**" instead.)

Here's the code for that part of the process:

```
LineInStr    DB "line in"

      CALL   waveInGetNumDevs          ;How many input devices?
      TEST   EAX, EAX                  ;Find any?
      JZ     noDevs                    ;  Nope, so that's an error.
      MOV    ECX, EAX                  ;Loop count = # input devs.
      MOV    devNum, 0

nxtIn: PUSH   ECX
      INVOKE waveInGetDevCaps, devNum, ADDR wic, SIZEOF WAVEINCAPS
      INVOKE strnicmp, ADDR wic.szPname, SIZEOF LineInStr, OFFSET LineInStr
      POP    ECX
      TEST   EAX, EAX                  ;Found it?
      JNZ    gotIn                     ; Yep!
      INC    devNum
      LOOP   nxtIn                     ;Look at next device, if any.
```

To know what's going on here you need to take a peek inside the **WAVEINCAPS** structure:

```
typedef struct {
   WORD    wMid;
   WORD    wPid;
   VERSION vDriverVersion;
   char    szPname[MAXPNAMELEN];
   DWORD   dwFormats;
   WORD    wChannels;
} WAVEINCAPS
```

The only element we're interested in here is **szPname**, which is a string (an array of bytes of maximum length **MAXPNAMELEN**) containing the name of the driver we tried to match above. If the beginning of that string matches "**LINE IN**" (case-insensitive), then we've found the driver we want to use.

OK, now that we've found the right driver (**devNum** will be set to the ID of the device), let's open it. First we need to set up a **WAVEFORMATEX** structure which is used to specify the settings we want for the device (the structure variable here is **wfx**):

```
MOV    wfx.wFormatTag, WAVE_FORMAT_PCM
MOV    wfx.nChannels, 1
MOV    wfx.nSamplesPerSec, 44100
MOV    wfx.nAvgBytesPerSec, 88200
MOV    wfx.nBlockAlign, 2
MOV    wfx.wBitsPerSample, 16
MOV    wfx.cbSize, 0
```

Be sure to set the **wFormatTag** element to **WAVE_FORMAT_PCM**;
**nBlockAlign** gets set to the # of bits/sample times the number of channels divided by 8;
**nAvgBytesPerSec** gets set to the number of samples/sec. multiplied by **nBlockAlign**.

Now we can open the device:

```
INVOKE waveInOpen, ADDR waveInDevHandle, devNum, ADDR wfx, 0, 0,
       WAVE_FORMAT_DIRECT
CMP    EAX, MMSYSERR_NOERROR
JE     @F
MOV    EDX, OFFSET WaveOpenErrMsg
JMP    errxit
```

Notice that the first parameter is the *address* of the device handle (**waveInDevHandle**) that's returned when the device is successfully opened. We need this handle for all other calls to the Wave device.

If the call succeeds, the return value is **MMSYSERR_NOERROR**. Otherwise, it's any of a number of error codes (you can look these up on the Microsoft Learn documentation site). All the Wave audio functions work this same way.

Now that the driver's at our disposal, let's make a recording. First we'll need to set up some buffers and do some buffer management; this is the trickiest part of the process.

Before we get to that code, let's look at how buffers are handled during recording.

When you record, you always need to use at least two buffers to store the audio data. Why? Because obviously you can't pause the recording, process some data, then restart the recording (well, usually you can't do that), because you'll lose some of the incoming data while you do that processing. So with two (or more) buffers, one buffer is always "recording", receiving data through the sound card, while you process the other one (examine the data, copy it to disk, whatever).

The thing to understand is that these two buffers—the active "recording" one and the one being processed—exist in two different states:

- The buffer recording and receiving the data is **locked** and cannot be accessed by you;
- The buffer that was done recording and is being processed is **unlocked** so that you can access it.

The locking and unlocking of the buffers, as well as other stuff internal to the Wave device, is handled by two special functions:

- **`waveInPrepareHeader()`** takes a buffer, locks it and lets the driver know that it's ready for recording
- **`waveInUnprepareHeader()`** unlocks a buffer that has been filled by the driver so you can access it

The names that Microsoft gave these two functions are unfortunate and confusing; if it were up to me I would have named them something like "**`waveInLockBufferForRecording()`**" and "**`waveInUnlockBuffer()`**", since that's what they actually do. But nobody asked me …

There's a third function needed for recording:

- **`waveInAddBuffer()`** adds a buffer to the list of active buffers being used by the device

While recording, the device driver maintains a list of active buffers. When you start recording, it starts filling the first buffer with data. When that buffer is filled, it notifies you that the buffer is full; see below for how that works. It then starts filling the *next* buffer in the list, and the process continues until the recording process is stopped.

You need to keep the ball rolling by adding "new" buffers to the list any time a buffer gets filled. With just two buffers you keep "adding" the other buffer—the one you just got done processing—as the next "new" one; do you see how this circular process works? It's like the method of moving a heavy object on rollers where the "old" rollers are taken from behind the thing being moved and put back in front, as if there were an infinite supply of rollers …

Anyhow, here's the sequence for recording, using the two buffers for our example:

- **1. Set up both buffers for recording:**
  - Allocate the buffers (I use **`HeapAlloc()`** for this). When sizing the buffers it's helpful to measure them in **seconds of audio recording**. If your sampling frequency is 44.1kHz and you're using 16-bit samples in mono (1 channel), then you need 88.2 kB (decimal) per second.
  - Create two **`WAVEHDR`** structures, one for each buffer. Plug in the needed info into these structs: the address of the buffer, its size and a bit more info (see code below).
    For each buffer, call **`waveInPrepareHeader()`** and then **`waveInAddBuffer()`**, using its respective **`WAVEHDR`** struct.
    In this example I've set up two variables, **`CurrRecHdrPtr`** and **`NextRecHdrPtr`**, each of which points to one of the **`WAVEHDR`** structures. You'll see later how these are used to keep track of the buffer-swapping process.
- **2. Start the recording**, using **`waveInStart()`**:
  The device driver will start filling the first buffer with audio data.

- **3. Wait until the first buffer is filled:**
  To do this, look at the **WAVEHDR** structure for the first buffer. In that struct, the **dwFlags** element contains a **WHDR_DONE** bit, which gets set when the buffer is filled. In this example we're using **polling**, meaning we just loop until this bit gets set; in the meantime, the device driver continues filling the buffer.

- **4. When the first buffer is filled, process it, then line up the next buffer to add:**
  When the **WHDR_DONE** flag gets set, unlock the buffer with **waveInUnprepareHeader()**. You can now process this data; in our example, it gets copied to a larger buffer which will be written to disk as a .wav file when the recording is done.
  After the buffer data has been processed, we need to set up this buffer as the next buffer for the device driver to fill after the one being currently recorded. Do this by calling **waveInPrepareHeader()** using the current structure pointed to by **CurrRecHdrPtr**, followed by **waveInAddBuffer()** to add this buffer as the "next" one.
  Last, we need to swap the current and "next" buffer pointers, **CurrRecHdrPtr** and **NextRecHdrPtr**, so that when the buffer now being filled gets full we can do the same thing we just did with the first buffer. Rinse and repeat …

These last two steps get repeated in a loop. In our example the process ends when the user indicates they want to stop the recording (by clicking a "Stop" button in the demo program), so in this loop we need to check the state of a flag that's set by the user interface when the user clicks that button. When the flag variable **Recording** is set to **FALSE**, we exit the loop, call **waveInStop()** to stop the recording, write the .wav file to disk and that's that. (Well, when the recording is stopped we need to check the active recording buffer and copy any remaining data out of it, as shown in the code below.)

Be aware that the code that processes the audio data is time-sensitive: if you take too long while the other buffer is filling, you risk losing some of that later data. You must finish processing the buffer before the other buffer fills up.

## Polling vs. Notification for Buffer-Full Events

Some of you reading this might not like the idea of polling, by looping to wait until a buffer is filled. It's true that this can result in high CPU usage for the program, and it can make the user interface unresponsive in a Windows GUI program. If you do use polling, then you definitely should put the recording code in a separate thread and set up some kind of simple protocol to communicate with the rest of the program. In my demo program I simply used global variables for this, which worked fine.

If you don't want to use polling, there are two other notification methods you can use with Waveform Audio to let your program know when a buffer is filled. Both of these methods are selected when calling **waveInOpen()**:

- **Using a window for callback notification:**
  By setting the **dwCallback** parameter to a window handle, and setting the **CALLBACK_WINDOW** flag in the **fdwOpen** parameter, a message (**MM_WIM_DATA**) will be sent to that window when a buffer becomes full.

- **Using a callback function:**
  You can create a special callback function that gets notified when a buffer becomes full. To do this you set the **dwCallback** parameter to the address of the function, and set the **CALLBACK_FUNCTION** flag in the **fdwOpen** parameter.
  However, don't even bother trying to use this method, because according to Microsoft's own documentation:

  > Applications should not call any system-defined functions from inside a callback function, except for EnterCriticalSection, LeaveCriticalSection, midiOutLongMsg, midiOutShortMsg, OutputDebugString, PostMessage, PostThreadMessage, SetEvent, timeGetSystemTime, timeGetTime, timeKillEvent, and timeSetEvent. Calling other wave functions will cause deadlock.

So either use polling in a separate thread, or use a window to receive the **MM_WIM_DATA** message.

If you're not using either of these notification methods, be sure to set **dwCallback** to **NULL** and don't set either the **CALLBACK_WINDOW** or **CALLBACK_FUNCTION** flags when opening the Wave device with **waveInOpen()**. (Just set the **fdwOpen** parameter to 0.)

Here's the rest of the recording code:

```
WaveBuffer1         DD ?                ;Allocated by HeapAlloc().
WaveBuffer2         DD ?

CurrRecHdrPtr       DD ?
NextRecHdrPtr       DD ?

Wh1                 WAVEHDR <>
Wh2                 WAVEHDR <>

; Prep the buffers and add them to start:
; Set up WAVEHDR struct.:
       MOV    EAX, WaveBuffer1
       MOV    Wh1.lpData, EAX
       MOV    Wh1.dwUser, EAX           ;This is for our use later.
       MOV    wh1.dwBufferLength, $waveBuffSize
       MOV    wh1.dwBytesRecorded, 0
       MOV    wh1.dwFlags, 0

       INVOKE waveInPrepareHeader, waveInDevHandle, ADDR Wh1, SIZEOF WAVEHDR
       CMP    EAX, MMSYSERR_NOERROR
       JE     @F
       MOV    EDX, OFFSET WavePrepBuffErrMsg
       JMP    errxit

@@:    INVOKE waveInAddBuffer, waveInDevHandle, ADDR Wh1, SIZEOF WAVEHDR
       CMP    EAX, MMSYSERR_NOERROR
       JE     @F
       MOV    EDX, OFFSET WaveAddBuffErrMsg
       JMP    errxit

@@:    MOV    EAX, WaveBuffer2
       MOV    Wh2.lpData, EAX
       MOV    Wh2.dwUser, EAX
       MOV    Wh2.dwBufferLength, $waveBuffSize
       MOV    Wh2.dwBytesRecorded, 0
       MOV    Wh2.dwFlags, 0

       INVOKE waveInPrepareHeader, waveInDevHandle, ADDR Wh2, SIZEOF WAVEHDR
       CMP    EAX, MMSYSERR_NOERROR
       JE     @F
       MOV    EDX, OFFSET WavePrepBuffErrMsg
       JMP    errxit

@@:    INVOKE waveInAddBuffer, waveInDevHandle, ADDR Wh2, SIZEOF WAVEHDR
       CMP    EAX, MMSYSERR_NOERROR
       JE     @F
       MOV    EDX, OFFSET WaveAddBuffErrMsg
       JMP    errxit

; Start the recording:
       LEA    EAX, Wh1
       MOV    CurrRecHdrPtr, EAX
       LEA    EAX, Wh2
       MOV    NextRecHdrPtr, EAX

       MOV    EAX, OutputBuffer
       MOV    OutputBufferPtr, EAX
       MOV    OutputBufferCount, 0

       INVOKE waveInStart, waveInDevHandle
```

```
continueRecording:
; This is where we poll the driver to see if it's done with the current buffer
       MOV    EDX, CurrRecHdrPtr

bufferLoop:
       CMP    Recording, 0                     ;Did user stop recording?
       JE     recordingDone
       TEST   [EDX].WAVEHDR.dwFlags, WHDR_DONE ;Is buffer full yet?
       JZ     bufferLoop

; "unprepare" buffer so we can get at it:
       INVOKE waveInUnprepareHeader, waveInDevHandle, EDX, SIZEOF WAVEHDR
       CMP    EAX, MMSYSERR_NOERROR
       JE     @F
       MOV    EDX, OFFSET WaveUnprepBuffErrMsg
       JMP    errxit

; Copy the buffer to the output buffer:
@@:    MOV    EDX, CurrRecHdrPtr
       MOV    ESI, [EDX].WAVEHDR.dwUser        ;Buffer address we stashed earlier.
       MOV    EDI, OutputBufferPtr
       MOV    ECX, [EDX].WAVEHDR.dwBytesRecorded
       ADD    OutputBufferPtr, ECX
       ADD    OutputBufferCount, ECX
       SHR    ECX, 1                           ;/ 2 for # of WORDs.
       REP    MOVSW

; Add "next" buffer to driver:
       MOV    EDX, NextRecHdrPtr
       MOV    EAX, [EDX].WAVEHDR.dwUser
       MOV    [EDX].WAVEHDR.lpData, EAX
       MOV    [EDX].WAVEHDR.dwBufferLength, $waveBuffSize
       MOV    [EDX].WAVEHDR.dwBytesRecorded, 0
       MOV    [EDX].WAVEHDR.dwFlags, 0         ;Be sure to clear this flag!

       INVOKE waveInPrepareHeader, waveInDevHandle, nextRecHdr, SIZEOF WAVEHDR
       CMP    EAX, MMSYSERR_NOERROR
       JE     @F
       MOV    EDX, OFFSET WavePrepBuffErrMsg
       JMP    errxit

@@:    INVOKE waveInAddBuffer, waveInDevHandle, nextRecHdr, SIZEOF WAVEHDR
       CMP    EAX, MMSYSERR_NOERROR
       JE     @F
       MOV    EDX, OFFSET WaveAddBuffErrMsg
       JMP    errxit

; Swap buffers:
@@:    MOV    EAX, CurrRecHdrPtr
       XCHG   EAX, NextRecHdrPtr
       MOV    CurrRecHdrPtr, EAX
       JMP    continueRecording
```

```
recordingDone:
       INVOKE waveInStop, waveInDevHandle

; Check for last buffer-full:
       INVOKE waveInUnprepareHeader, waveInDevHandle, CurrRecHdrPtr, SIZEOF WAVEHDR
       CMP    EAX, MMSYSERR_NOERROR
       JE     @F
       MOV    EDX, OFFSET WaveUnprepBuffErrMsg
       JMP    errxit

@@:    MOV    EDX, CurrRecHdrPtr
       MOV    ECX, [EDX].WAVEHDR.dwBytesRecorded
       JECXZ  @F

       MOV    ESI, [EDX].WAVEHDR.dwUser
       MOV    EDI, OutputBufferPtr
       ADD    OutputBufferPtr, ECX
       ADD    OutputBufferCount, ECX
       SHR    ECX, 1                          ;/ 2 for # of WORDs.
       REP    MOVSW

; Write output buffer to WAV file:
@@:    CALL   WriteWaveFile
       INVOKE waveInClose, waveInDevHandle
       JMP    exit99                          ;Exit thread.
```