Multimedia API (Waveform and Auxiliary Audio)

- We will look at some of the functions for recording and playing **waveform audio** signals.

- Windows provides many functions for working with waveform-audio and auxiliary devices as part of the **multimedia API**s.

- The **waveform API**s are separated into three groups:

    - General purpose playback functions
    - Low-level playback functions
    - Low-level recording functions

Waveform Audio

- Waveform audio is a method of recording and playing back **digitally sampled** sound.

- The Windows Multimedia API supports a range of **sampling rates** from **11.025 kHz** to **44 kHz**.

- The Windows Multimedia API supports two **digital resolutions**: **8-bit** and **16-bit**.

- Note that the storage requirements double for stereo waveform audio because stereo audio requires two samples (left and right channels) for each sampling interval.

General Purpose Waveform Audio

- The general purpose playback function **PlaySound()** can be used for playback of waveform audio from various sources such as **.WAV** files.

- The function plays a sound specified by the given filename, resource, or system event. (A system event may be associated with a sound in the registry or in the WIN.INI file.):

    BOOL PlaySound(LPCSTR pszSound, HMODULE hmod, DWORD fdwSound);

- Where

    pszSound

    A string that specifies the sound to play.

    hmod

    Handle of the executable file that contains the resource to be loaded. This parameter must be NULL unless SND_RESOURCE is specified in fdwSound.

fdwSound

Flags for playing the sound.

- You can use this function to play sounds continuously in a **synchronous** or an **asynchronous** loop.
- Synchronous playback returns after the sound segment has been played.
- Asynchronous playback returns immediately and plays the sound in the background.
- This is quick and easy way to play a .WAV audio file but it can lock up the console (frozen mouse and keyboard) since it loads a large audio file into memory for playback.
- This problem can be overcome by calling this function from a separate thread, or by using Low-level waveform audio.
- The following is an example high-level function to play a local file on the locak machine.

```
/*****************************************************************************
 * Function Name:  PlayLocalAudioFile -- plays a local audio file
 *
 * Interface:    int PlayLocalAudioFile(char *filename);
 *
 * ARGUMENTS
 *    filename - name of audio file to be played
 * RETURN VALUE
 *    TRUE - successful, FALSE - otherwise
 *****************************************************************************/
int PlayLocalAudioFile(char *filename)
{
        StopPlayback();

        PlaySound(filename, hInst, SND_FILENAME | SND_ASYNC);
        MessageBox ( hMainWnd,
            "Press OK button to stop playback and return to main window.",
            "Playing...", MB_OK);
        StopPlayback();

        return TRUE;
}
```

- We can use the same function to stop playback as shown below:

```
/*****************************************************************************
 * Function name:  StopPlayback -- stop playback of audio file
 * Interface:   void StopPlayback(void);
 * ARGUMENTS
 *    None
 * RETURN VALUE
 *    None
 *****************************************************************************/
void StopPlayback (void)
{
        PlaySound (NULL, hInst, SND_PURGE);
}
```

Low_level Waveform Audio

- These functions allow us to exercise more **control** over the playback, something we cannot do using the high-level **PlaySound()** function.
- The application is now required to supply source and destination device parameters, control looping, volume, waveform audio formats selection, sample and playback rates, and pitch.
- The basic steps for low-level play or record are as follows:
    1. Open the desired device.
    2. Allocate and initialize one or more data buffer blocks to be used for passing waveform-audio sample data back and forth to the device.
    3. Place the data buffer blocks in the device's queue for processing.
- The sample code provides functions that illustrate the how to allocate and manage the data buffer blocks.
- Also shown is a function that can be used to free the data buffer blocks.
- The next example illustrates one method for opening a waveform output device for playback.
- The data buffer blocks are handed off to the device by calling the **waveOutWrite()** function.
- Each data block is played back in the order in which it was received.
- The **waveOutPause()** function can be used to load all the data blocks before playback begins. This function will pause the device and allow all the data blocks to be loaded before playback begins.

- Then, we use the **waveOutRestart()** function to commence playback.
- The **waveOutReset()** function can be used to clear all data blocks from the queue.
- We can specify a callback function to be called in the **waveOutOpen()** function which is used to open an output device.
- This callback function can be used to receive messages indicating the progress of the device. In the example given, the specified callback function is the **WndProc** function (**(DWORD)hWnd)**).
- One such message is the **MM_WOM_DONE** message which notifies the application when each data block has been played back.
- The **MM_WOM_DONE** message can then be used to post a use defined message (**USR_OUTBLOCK** for example) which can be used to reuses the data buffer block.
- Within that message case we can fill the data buffer block with more sound data and hand it off to the output device to be played.

- Similar functions exist for recording waveform audio.
- The first step is to locate a suitable waveform-audio recording device using the **waveInNumDevs()** function.
- The above function will determine the number of waveform-audio input devices currently installed.
- For each device, the **waveInGetDevCaps()** function can be used to retrieve the capabilities of the selected device into a **WAVEINCAPS** structure.
- The example provided illustrates one method for opening a waveform input device for recording.
- Once a device has been located and opened, the data buffer blocks must be allocated, prepared, and added to the device queue.
- Allocate the memory required for each data buffer block header (**WAVEHDR** structure), and then allocate the data block to which the header points.
- The pointer to the data block is placed in the **lpData WAVEHDR** member variable.
- Each data block can be uniquely identified by placing an identification tag, sequence number, or data structure in the **dwUser WAVEHDR** member variable.
- Allocated data buffer blocks must be prepared before they are used.
- The **waveInPrepareHeader()** function is used to prepare each data buffer block header.
- Then we use the **waveInAddBuffer()** function is then called to place the data buffer on the device input queue.

- The device takes data blocks off the queue in the order they were supplied and fills them with recorded waveform-audio data.
- The **waveInReset()** function can be used to clear all data blocks from the queue.
- After each data block has had recorded data placed in it by the waveform-audio input device, it is returned to the application for processing.
- These data blocks can then be processed, prepared again, and placed back into the input queue to receive freshly recorded sound data.
- Each data buffer block must be unprepared by calling the **waveInUnprepareHeader()** function before it can be freed and disposed of.
- A callback function can be specified in the waveInOpen() function when the waveform-audio input device is opened.
- This callback function can be used to receive messages indicating the progress of the device.
- In the example given, the specified callback function is the **waveInProc()** function is to be used as the callback function .
- Note the use of the **WIM_DATA** message in **waveInProc()** which notifies the application that a data block has been filled with recorded audio and requires processing.

```
VOID CALLBACK waveInProc( HWAVEIN hwi, UINT uMsg, DWORD dwInstance,
DWORD dwParam1, DWORD dwParam2 )
{
  switch (uMsg)
  {
    case WIM_OPEN :
    case WIM_CLOSE:
          break;  // don't care

    case WIM_DATA :
          {
            // post message to process this input block received
            // NOTE: callback cannot call other waveform functions
            //.....

            PostMessage((HWND)dwInstance, USR_INBLOCK, 0, dwParam1);

            break;
          }
  }
}
```

- The above example will monitor the data blocks received from the waveform-audio input device and passes them to the main **WndProc()**.

- Note that calling other waveform functions in a callback function can lock up the system.

- Functions that cannot be executed quickly should not be called while processing a waveform callback message as this could cause audio gaps in the recorded waveform stream.

- To solve this problem, a message is posted to instruct the application to perform the required processing.

Media Control Interface (MCI)

- The **Media Control Interface** (**MCI**) is a high-level command interface to multimedia devices and resource files.
- MCI provides applications with device-independent capabilities for controlling audio and visual peripherals. Your application can use MCI to control any supported multimedia device, including waveform-audio devices, MIDI sequencers, CD audio devices, and digital-video (video playback) devices.
- MCI provides standard commands for playing multimedia devices and recording multimedia resource files. These commands are a **generic interface** to virtually every kind of multimedia device.
- There are two forms of MCI commands: **strings** and **messages**. You can use either or both forms in your MCI application.
- The **command-message** interface consists of constants and structures of the C programming language. You use the **mciSendCommand** function to send a message to an MCI device.
- The **command-string** interface provides a textual version of the command messages. You use the **mciSendString** function to send a string to an MCI device.
- Command strings duplicate the functionality of the command messages. The Windows operating system converts the command strings to command messages before sending them to the MCI driver for processing.
- The command messages that retrieve information do so in the form of structures, which are easy to interpret in a C application.
- These structures can contain information on many different aspects of a device. The command strings that retrieve information do so in the form of strings, and can only retrieve one string at a time.
- Your application must parse or test each string to interpret it. You might find that the command messages are easier to use than the command strings in some cases, but the command strings are easy to remember and implement.
- Some MCI applications use command strings when the return value is unimportant or simply "true" or "false" and command messages when retrieving information from the device.
- The example functions provided are some examples of MCI audio output routines.
- Also provided are examples of MCI audio input routines.