Exclusive-Mode Streams (Windows)

3 out of 3 rated this helpful

As explained previously, if an application opens a stream in exclusive mode, the application has exclusive use of the audio endpoint device that plays or records the stream. In contrast, several applications can share an audio endpoint device by opening shared-mode streams on the device.

Exclusive-mode access to an audio device can block crucial system sounds, prevent interoperability with other applications, and otherwise degrade the user experience. To mitigate these problems, an application with an exclusive-mode stream typically relinquishes control of the audio device when the application is not the foreground process or is not actively streaming.

Stream latency is the delay that is inherent in the data path that connects an application's endpoint buffer with an audio endpoint device. For a rendering stream, the latency is the maximum delay from the time that an application writes a sample to an endpoint buffer to the time that the sample is heard through the speakers. For a capture stream, the latency is the maximum delay from the time that a sound enters the microphone to the time that an application can read the sample for that sound from the endpoint buffer.

Applications that use exclusive-mode streams often do so because they require low latencies in the data paths between the audio endpoint devices and the application threads that access the endpoint buffers. Typically, these threads run at relatively high priority and schedule themselves to run at periodic intervals that are close to or the same as the periodic interval that separates successive processing passes by the audio hardware. During each pass, the audio hardware processes the new data in the endpoint buffers.

To achieve the smallest stream latencies, an application might require both special audio hardware, and a computer system that is lightly loaded. Driving the audio hardware beyond its timing limits or loading the system with competing high-priority tasks might cause a glitch in a low-latency audio stream. For example, for a rendering stream, a glitch can occur if the application fails to write to an endpoint buffer before the audio hardware reads the buffer, or if the hardware fails to read the buffer before the time that the buffer is scheduled to play. Typically, an application that is intended to run on a wide variety of audio hardware and in a broad range of systems should relax its timing requirements enough to avoid glitches in all target environments.

Windows Vista has several features to support applications that require low-latency audio streams. As discussed in User-Mode Audio Components, applications that perform time-critical operations can call the Multimedia Class Scheduler Service (MMCSS) functions to increase thread priority without denying CPU resources to lower-priority applications. In addition, the **IAudioClient::Initialize** method supports an AUDCLNT_STREAMFLAGS_EVENTCALLBACK flag that enables an application's buffer-servicing thread to schedule its execution to occur when a new buffer becomes available from the audio device. By using these features, an application thread can reduce uncertainty about when it will execute, thereby decreasing the risk of glitches in a low-latency audio stream.

The drivers for older audio adapters are likely to use the WaveCyclic or WavePci device driver interface (DDI), whereas the drivers for newer audio adapters are more likely to support the WaveRT DDI. For exclusive-mode applications, WaveRT drivers can provide better performance than WaveCyclic or WavePci drivers, but WaveRT drivers require additional hardware capabilities. These capabilities include the ability to share hardware buffers directly with applications. With direct sharing, no system intervention is required to transfer

data between an exclusive-mode application and the audio hardware. In contrast, WaveCyclic and WavePci drivers are suitable for older, less capable audio adapters. These adapters rely on system software to transport blocks of data (attached to system I/O request packets, or IRPs) between application buffers and hardware buffers. In addition, USB audio devices rely on system software to transport data between application buffers and hardware buffers. To improve the performance of exclusive-mode applications that connect to audio devices that rely on the system for data transport, WASAPI automatically increases the priority of the system threads that transfer data between the applications and the hardware. WASAPI uses MMCSS to increase the thread priority. In Windows Vista, if a system thread manages data transport for an exclusive-mode audio playback stream with a PCM format and a device period of less than 10 milliseconds, WASAPI assigns the MMCSS task name "Pro Audio" to the thread. If the device period of the stream is greater than or equal to 10 milliseconds, WASAPI assigns the MMCSS task name "Audio" to the thread. For more information about the WaveCyclic, WavePci, and WaveRT DDIs, see the Windows DDK documentation. For information about selecting an appropriate device period, see **IAudioClient::GetDevicePeriod**.

As described in Session Volume Controls, WASAPI provides the **ISimpleAudioVolume**, **IChannelAudioVolume**, and **IAudioStreamVolume** interfaces for controlling the volume levels of shared-mode audio streams. However, the controls in these interfaces have no effect on exclusive-mode streams. Instead, applications that manage exclusive-mode streams typically use the **IAudioEndpointVolume** interface in the EndpointVolume API to control the volume levels of those streams. For information about this interface, see Endpoint Volume Controls.

For each playback device and capture device in the system, the user can control whether the device can be used in exclusive mode. If the user disables exclusive-mode use of the device, the device can be used to play or record only shared-mode streams.

If the user enables exclusive-mode use of the device, the user can also control whether a request by an application to use the device in exclusive mode will preempt the use of the device by applications that might currently be playing or recording shared-mode streams through the device. If preemption is enabled, a request by an application to take exclusive control of the device succeeds if the device is currently not in use, or if the device is being used in shared mode, but the request fails if another application already has exclusive control of the device. If preemption is disabled, a request by an application to take exclusive control of the device succeeds if the device is not currently in use, but the request fails if the device is already being used either in shared mode or in exclusive mode.

In Windows Vista, the default settings for an audio endpoint device are the following:

- The device can be used to play or record exclusive-mode streams.
- A request to use a device to play or record an exclusive-mode stream preempts any shared-mode stream that is currently being played or recorded through the device.

To change the exclusive-mode settings of a playback or recording device

1. Right-click the speaker icon in the notification area, which is located on the right side of the taskbar, and select **Playback Devices** or **Recording Devices**. (As an alternative, run the Windows multimedia control panel, Mmsys.cpl, from a Command Prompt window. For more information, see Remarks in DEVICE_STATE_XXX Constants.)
2. After the **Sound** window appears, select **Playback** or **Recording**. Next, select an entry in the list of device names, and click **Properties**.
3. After the **Properties** window appears, click **Advanced**.
4. To enable applications to use the device in exclusive mode, check the box labeled **Allow applications to take exclusive control of this device**. To disable exclusive-mode use of the device, clear the check

box.

5. If exclusive-mode use of the device is enabled, you can specify whether a request for exclusive control of the device will succeed if the device is currently playing or recording shared-mode streams. To give exclusive-mode applications priority over shared-mode applications, check the box labeled **Give exclusive mode applications priority**. To deny exclusive-mode applications priority over shared-mode applications, clear the check box.

The following code example shows how to play a low-latency audio stream on an audio-rendering device that is configured for use in exclusive mode:

```
//-----------------------------------------------------------
// Play an exclusive-mode stream on the default audio
// rendering device. The PlayExclusiveStream function uses
// event-driven buffering and MMCSS to play the stream at
// the minimum latency supported by the device.
//-----------------------------------------------------------

// REFERENCE_TIME time units per second and per millisecond
#define REFTIMES_PER_SEC  10000000
#define REFTIMES_PER_MILLISEC  10000

#define EXIT_ON_ERROR(hres)  \
              if (FAILED(hres)) { goto Exit; }
#define SAFE_RELEASE(punk)  \
              if ((punk) != NULL)  \
                { (punk)->Release(); (punk) = NULL; }

const CLSID CLSID_MMDeviceEnumerator = __uuidof(MMDeviceEnumerator);
const IID IID_IMMDeviceEnumerator = __uuidof(IMMDeviceEnumerator);
const IID IID_IAudioClient = __uuidof(IAudioClient);
const IID IID_IAudioRenderClient = __uuidof(IAudioRenderClient);

HRESULT PlayExclusiveStream(MyAudioSource *pMySource)
{
    HRESULT hr;
    REFERENCE_TIME hnsRequestedDuration = 0;
    IMMDeviceEnumerator *pEnumerator = NULL;
    IMMDevice *pDevice = NULL;
    IAudioClient *pAudioClient = NULL;
    IAudioRenderClient *pRenderClient = NULL;
    WAVEFORMATEX *pwfx = NULL;
    HANDLE hEvent = NULL;
    HANDLE hTask = NULL;
    UINT32 bufferFrameCount;
    BYTE *pData;
    DWORD flags = 0;
```

```
hr = CoCreateInstance(
       CLSID_MMDeviceEnumerator, NULL,
       CLSCTX_ALL, IID_IMMDeviceEnumerator,
       (void**)&pEnumerator);
EXIT_ON_ERROR(hr)

hr = pEnumerator->GetDefaultAudioEndpoint(
                    eRender, eConsole, &pDevice);
EXIT_ON_ERROR(hr)

hr = pDevice->Activate(
                IID_IAudioClient, CLSCTX_ALL,
                NULL, (void**)&pAudioClient);
EXIT_ON_ERROR(hr)

// Call a helper function to negotiate with the audio
// device for an exclusive-mode stream format.
hr = GetStreamFormat(pAudioClient, &pwfx);
EXIT_ON_ERROR(hr)

// Initialize the stream to play at the minimum latency.
hr = pAudioClient->GetDevicePeriod(NULL, &hnsRequestedDuration);
EXIT_ON_ERROR(hr)

hr = pAudioClient->Initialize(
                     AUDCLNT_SHAREMODE_EXCLUSIVE,
                     AUDCLNT_STREAMFLAGS_EVENTCALLBACK,
                     hnsRequestedDuration,
                     hnsRequestedDuration,
                     pwfx,
                     NULL);
EXIT_ON_ERROR(hr)

// Tell the audio source which format to use.
hr = pMySource->SetFormat(pwfx);
EXIT_ON_ERROR(hr)

// Create an event handle and register it for
// buffer-event notifications.
hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
if (hEvent == NULL)
{
    hr = E_FAIL;
    goto Exit;
}

hr = pAudioClient->SetEventHandle(hEvent);
EXIT_ON_ERROR(hr);

// Get the actual size of the two allocated buffers.
hr = pAudioClient->GetBufferSize(&bufferFrameCount);
```

```
    EXIT_ON_ERROR(hr)

    hr = pAudioClient->GetService(
                         IID_IAudioRenderClient,
                         (void**)&pRenderClient);
    EXIT_ON_ERROR(hr)

    // To reduce latency, load the first buffer with data
    // from the audio source before starting the stream.
    hr = pRenderClient->GetBuffer(bufferFrameCount, &pData);
    EXIT_ON_ERROR(hr)

    hr = pMySource->LoadData(bufferFrameCount, pData, &flags);
    EXIT_ON_ERROR(hr)

    hr = pRenderClient->ReleaseBuffer(bufferFrameCount, flags);
    EXIT_ON_ERROR(hr)

    // Ask MMCSS to temporarily boost the thread priority
    // to reduce glitches while the low-latency stream plays.
    DWORD taskIndex = 0;
    hTask = AvSetMmThreadCharacteristics(TEXT("Pro Audio"), &taskIndex);
    if (hTask == NULL)
    {
        hr = E_FAIL;
        EXIT_ON_ERROR(hr)
    }

    hr = pAudioClient->Start();  // Start playing.
    EXIT_ON_ERROR(hr)

    // Each loop fills one of the two buffers.
    while (flags != AUDCLNT_BUFFERFLAGS_SILENT)
    {
        // Wait for next buffer event to be signaled.
        DWORD retval = WaitForSingleObject(hEvent, 2000);
        if (retval != WAIT_OBJECT_0)
        {
            // Event handle timed out after a 2-second wait.
            pAudioClient->Stop();
            hr = ERROR_TIMEOUT;
            goto Exit;
        }

        // Grab the next empty buffer from the audio device.
        hr = pRenderClient->GetBuffer(bufferFrameCount, &pData);
        EXIT_ON_ERROR(hr)

        // Load the buffer with data from the audio source.
        hr = pMySource->LoadData(bufferFrameCount, pData, &flags);
        EXIT_ON_ERROR(hr)
```

```
        hr = pRenderClient->ReleaseBuffer(bufferFrameCount, flags);
        EXIT_ON_ERROR(hr)
    }

    // Wait for the last buffer to play before stopping.
    Sleep((DWORD)(hnsRequestedDuration/REFTIMES_PER_MILLISEC));

    hr = pAudioClient->Stop();  // Stop playing.
    EXIT_ON_ERROR(hr)

Exit:
    if (hEvent != NULL)
    {
        CloseHandle(hEvent);
    }
    if (hTask != NULL)
    {
        AvRevertMmThreadCharacteristics(hTask);
    }
    CoTaskMemFree(pwfx);
    SAFE_RELEASE(pEnumerator)
    SAFE_RELEASE(pDevice)
    SAFE_RELEASE(pAudioClient)
    SAFE_RELEASE(pRenderClient)

    return hr;
}
```

In the preceding code example, the PlayExclusiveStream function runs in the application thread that services the endpoint buffers while a rendering stream is playing. The function takes a single parameter, pMySource, which is a pointer to an object that belongs to a client-defined class, MyAudioSource. This class has two member functions, LoadData and SetFormat, that are called in the code example. MyAudioSource is described in Rendering a Stream.

The PlayExclusiveStream function calls a helper function, GetStreamFormat, that negotiates with the default rendering device to determine whether the device supports an exclusive-mode stream format that is suitable for use by the application. The code for the GetStreamFormat function does not appear in the code example; that is because the details of its implementation depend entirely on the requirements of the application. However, the operation of the GetStreamFormat function can be described simply—it calls the **IAudioClient::IsFormatSupported** method one or more times to determine whether the device supports a suitable format. The requirements of the application dictate which formats GetStreamFormat presents to the **IsFormatSupported** method and the order in which it presents them. For more information about **IsFormatSupported**, see Device Formats.

After the GetStreamFormat call, the PlayExclusiveStream function calls the **IAudioClient::GetDevicePeriod** method to obtain the minimum device period supported by the audio hardware. Next, the function calls the **IAudioClient::Initialize** method to request a buffer duration equal to the minimum period. If the call

succeeds, the **Initialize** method allocates two endpoint buffers, each of which is equal in duration to the minimum period. Later, when the audio stream begins running, the application and audio hardware will share the two buffers in "ping-pong" fashion—that is, while the application writes to one buffer, the hardware reads from the other buffer.

Before starting the stream, the PlayExclusiveStream function does the following:

- Creates and registers the event handle through which it will receive notifications when buffers become ready to fill.
- Fills the first buffer with data from the audio source to reduce the delay from when the stream starts running to when the initial sound is heard.
- Calls the **AvSetMmThreadCharacteristics** function to request that MMCSS increase the priority of the thread in which PlayExclusiveStream executes. (When the stream stops running, the **AvRevertMmThreadCharacteristics** function call restores the original thread priority.)

For more information about **AvSetMmThreadCharacteristics** and **AvRevertMmThreadCharacteristics**, see the Windows SDK documentation.

While the stream is running, each iteration of the **while**-loop in the preceding code example fills one endpoint buffer. Between iterations, the **WaitForSingleObject** function call waits for the event handle to be signaled. When the handle is signaled, the loop body does the following:

1. Calls the **IAudioRenderClient::GetBuffer** method to get the next buffer.
2. Fills the buffer.
3. Calls the **IAudioRenderClient::ReleaseBuffer** method to release the buffer.

For more information about **WaitForSingleObject**, see the Windows SDK documentation.

If the audio adapter is controlled by a WaveRT driver, the signaling of the event handle is tied to the DMA-transfer notifications from the audio hardware. For a USB audio device, or for an audio device that is controlled by a WaveCyclic or WavePci driver, the signaling of the event handle is tied to completions of the IRPs that transfer data from the application buffer to the hardware buffer.

The preceding code example pushes the audio hardware and the computer system to their performance limits. First, to reduce the stream latency, the application schedules its buffer-servicing thread to use the minimum device period that the audio hardware will support. Second, to ensure that the thread reliably executes within each device period, the **AvSetMmThreadCharacteristics** function call sets the TaskName parameter to "Pro Audio", which is, in Windows Vista, the default task name with the highest priority. Consider whether the timing requirements of your application might be relaxed without compromising its usefulness. For example, the application might schedule its buffer-servicing thread to use a period that is longer than the minimum. A longer period might safely allow the use of a lower thread priority.

Build date: 10/16/2012

Community Additions

A Resolution to Strange Problems ?

When using this sample in a simple console application, I was experiencing some sporadic and strange problems that seemed difficult to debug. This seemed to occur when running several renders in succession from a command file (.cmd).

I tried adding some time to the sleeps but it didnt seem to help. Attaching to processes before they failed and then debugging the crash, resulted in breaks to crt startup or, sometimes, in user written code -- with no obvious user code problem.

I changed the order of the cleanup and this seemed to clear it up. Generally the cleanup is being done in reverse order and the event is released after disposing the audio interfaces.

So far the problem has been cleared up. I havent had time to revert and watch to see if the problems occur again.

...

code fragment below

```
// Wait for the last buffer to play before stopping.
    Sleep((DWORD)(hnsRequestedDuration/REFTIMES_PER_MILLISEC) + 100); // make sure it is done

    hr = pAudioClient->Stop();  // Stop playing.

    EXIT_ON_ERROR(hr)

Exit:

    if (hTask != NULL)
    {
        AvRevertMmThreadCharacteristics(hTask);
    }

    delete cbuffer ;

    CoTaskMemFree(pwfxMix);

    CoTaskMemFree(pwfx_match);

    SAFE_RELEASE(pRenderClient)

    SAFE_RELEASE(pAudioClient)

    SAFE_RELEASE(pDevice)

    SAFE_RELEASE(pEnumerator)

if (hEvent != NULL)
    {
        CloseHandle(hEvent);
    }
```

Sleep(100); // make sure this is done a well

Z3514

10/30/2011