http://www.cnblogs.com/del/category/123296.html

WinAPI: waveOutWrite - 向输出设备发送一个数据块

提示: 把数据缓冲区传给 waveOutWrite 之前, 必须使用 waveOutPrepareHeader 准备该缓冲区;

若未调用 waveOutPause 函数暂停设备, 则第一次把数据块发送给设备时即开始播放.

```
//声明:
waveOutWrite(
  hWaveOut: HWAVEOUT;     {设备句柄}
  lpWaveOutHdr: PWaveHdr; {TWaveHdr 结构的指针}
  uSize: UINT             {TWaveHdr 结构大小}
): MMRESULT;              {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}
WAVERR_UNPREPARED    = 34; {未准备数据块}

//TWaveHdr 是 wavehdr_tag 结构的重定义
wavehdr_tag = record
  lpData: PChar;          {指向波形数据缓冲区}
  dwBufferLength: DWORD;  {波形数据缓冲区的长度}
  dwBytesRecorded: DWORD; {若首部用于输入, 指出缓冲区中的数据量}
  dwUser: DWORD;          {指定用户的 32 位数据}
  dwFlags: DWORD;         {缓冲区标志}
  dwLoops: DWORD;         {循环播放次数, 仅用于输出缓冲区}
  lpNext: PWaveHdr;       {保留}
  reserved: DWORD;        {保留}
end;

//TWaveHdr 中的 dwFlags 的可选值:
WHDR_DONE      = $00000001; {设备已使用完缓冲区, 并返回给程序}
WHDR_PREPARED  = $00000002; {waveInPrepareHeader 或 waveOutPrepareH
eader 已将缓冲区准备好}
```

```
WHDR_BEGINLOOP = $00000004; {缓冲区是循环中的第一个缓冲区, 仅用于输出}
WHDR_ENDLOOP   = $00000008; {缓冲区是循环中的最后一个缓冲区, 仅用于输出}
WHDR_INQUEUE   = $00000010; { reserved for driver }
```

```
//举例:
```

WinAPI: waveOutUnprepareHeader - 清除由 waveOutPrepareHeader 完成的准备

提示:

设备使用完数据块后, 须调用此函数;

释放(GlobalFree)缓冲区前, 须调用此函数;

取消一个尚未准备的缓冲区将无效, 但函数返回 0

```
//声明:
waveOutUnprepareHeader(
  hWaveOut: HWAVEOUT;     {设备句柄}
  lpWaveOutHdr: PWaveHdr; {TWaveHdr 结构的指针}
  uSize: UINT             {TWaveHdr 结构大小}
): MMRESULT;              {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}
WAVERR_STILLPLAYING  = 33; {缓冲区还在队列中}

//TWaveHdr 是 wavehdr_tag 结构的重定义
wavehdr_tag = record
  lpData: PChar;          {指向波形数据缓冲区}
  dwBufferLength: DWORD;  {波形数据缓冲区的长度}
  dwBytesRecorded: DWORD; {若首部用于输入, 指出缓冲区中的数据量}
  dwUser: DWORD;          {指定用户的 32 位数据}
  dwFlags: DWORD;         {缓冲区标志}
```

```
  dwLoops: DWORD;         {循环播放次数, 仅用于输出缓冲区}
  lpNext: PWaveHdr;       {保留}
  reserved: DWORD;        {保留}
end;

//TWaveHdr 中的 dwFlags 的可选值:
WHDR_DONE      = $00000001; {设备已使用完缓冲区, 并返回给程序}
WHDR_PREPARED  = $00000002; {waveInPrepareHeader 或 waveOutPrepareH
eader 已将缓冲区准备好}
WHDR_BEGINLOOP = $00000004; {缓冲区是循环中的第一个缓冲区, 仅用于输出}
WHDR_ENDLOOP   = $00000008; {缓冲区是循环中的最后一个缓冲区, 仅用于输出}
WHDR_INQUEUE   = $00000010; { reserved for driver }
```

```
//举例:
```

WinAPI: waveOutGetPlaybackRate - 设置输出设备的播放速度(默认速度值的倍数)

提示:

参数 dwRate 虽然是 4 字节的正整数, 但表示的是个小数;

两个高位表示整数部分, 两个低位表示小数部分;

$8000 表示一半, $4000 表示四分之一;

譬如: $00010000 表示 1.0, 说明速度没有改变变; $000F8000, 表示 15.5 倍;

修改播放速度不会改变采样速度, 但肯定会改变播放时间.

```
//声明:
waveOutSetPlaybackRate(
  hWaveOut: HWAVEOUT; {设备句柄}
  dwRate: DWORD       {存放速度值的变量}
): MMRESULT;          {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE  = 5;  {设备句柄无效}
```

```
MMSYSERR_NOTSUPPORTED = 8;  {设备不支持}
MMSYSERR_HANDLEBUSY   = 12; {设备已被另一线程使用}
```

```
//举例:
```

WinAPI: waveOutSetPitch - 设置输出设备的音调设置(音高的倍数值)

提示:

参数 dwPitch 虽然是 4 字节的正整数，但表示的是个小数;

两个高位表示整数部分，两个低位表示小数部分;

$8000 表示一半，$4000 表示四分之一;

譬如: $00010000 表示 1.0，说明音高没变; $000F8000，表示 15.5 倍;

修改音高不会改变播放速度、采样速度和播放时间，但不是所有设备都支持.

```
//声明:
waveOutSetPitch(
  hWaveOut: HWAVEOUT; {设备句柄}
  dwPitch: DWORD   {存放音高值的变量}
): MMRESULT;          {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE  = 5;  {设备句柄无效}
MMSYSERR_NOTSUPPORTED = 8;  {设备不支持}
MMSYSERR_HANDLEBUSY   = 12; {设备已被另一线程使用}
```

```
//举例:
```

WinAPI: waveOutRestart - 重新启动一个被暂停的输出设备

提示: 当输出设备未暂停时调用该函数无效，但也返回 0

```
//声明:
waveOutRestart(
  hWaveOut: HWAVEOUT {设备句柄}
): MMRESULT;          {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}
```

```
//举例:
```

WinAPI: waveOutReset - 重置输出

提示: 函数会终止输入, 位置清 0; 放弃未处理的缓冲区并返回给程序.

```
//声明:
waveOutReset(
  hWaveOut: HWAVEOUT {设备句柄}
): MMRESULT;          {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}
```

```
//举例:
```

WinAPI: waveOutPrepareHeader - 准备一个波形数据块用于播放

提示: 必须调用 GlobalAlloc 给 TWaveHdr 和其中的 lpData 指向的缓冲区分配内存(使用 GMEM_MOVEABLE、GMEM_SHARE), 并用 GlobalLock 锁定.

```
//声明:
waveOutPrepareHeader(
  hWaveOut: HWAVEOUT;     {设备句柄}
  lpWaveOutHdr: PWaveHdr; {TWaveHdr 结构的指针}
  uSize: UINT             {TWaveHdr 结构大小}
): MMRESULT;              {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_NOMEM      = 7;  {不能分配或锁定内存}
MMSYSERR_HANDLEBUSY  = 12; {其他线程正在使用该设备}

//TWaveHdr 是 wavehdr_tag 结构的重定义
wavehdr_tag = record
  lpData: PChar;          {指向波形数据缓冲区}
  dwBufferLength: DWORD;  {波形数据缓冲区的长度}
  dwBytesRecorded: DWORD; {若首部用于输入, 指出缓冲区中的数据量}
  dwUser: DWORD;          {指定用户的 32 位数据}
  dwFlags: DWORD;         {缓冲区标志}
  dwLoops: DWORD;         {循环播放次数, 仅用于输出缓冲区}
  lpNext: PWaveHdr;       {保留}
  reserved: DWORD;        {保留}
end;

//TWaveHdr 中的 dwFlags 的可选值:
WHDR_DONE      = $00000001; {设备已使用完缓冲区, 并返回给程序}
WHDR_PREPARED  = $00000002; {waveInPrepareHeader 或 waveOutPrepareH
eader 已将缓冲区准备好}
WHDR_BEGINLOOP = $00000004; {缓冲区是循环中的第一个缓冲区, 仅用于输出}
WHDR_ENDLOOP   = $00000008; {缓冲区是循环中的最后一个缓冲区, 仅用于输出}
WHDR_INQUEUE   = $00000010; { reserved for driver }
```

```
//举例:
```

WinAPI: waveOutPause - 暂停播放

提示: 暂停后会保存当前位置, 可以用 waveOutRestart 从当前位置恢复播放.

```
//声明:
waveOutPause(
  hWaveOut: HWAVEOUT {设备句柄}
): MMRESULT;          {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE  = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY   = 12; {设备已被另一线程使用}
```

```
//举例:
```

WinAPI: waveOutOpen - 打开波形输出设备

提示: 因为其中的回调函数是在中断时间内访问的, 必须在 DLL 中; 要访问的数据都必须是在固定的数据段中; 除了

PostMessage

timeGetSystemTime

timeGetTime

timeSetEvent

timeKillEvent

midiOutShortMsg

midiOutLongMsg

OutputDebugString 外, 也不能有其他系统调用.

```
//声明:
waveOutOpen(
  lphWaveOut: PHWaveOut;   {用于返回设备句柄的指针; 如果 dwFlags=WAVE_FORMAT_QUERY, 这里应是 nil}
  uDeviceID: UINT;         {设备 ID; 可以指定为: WAVE_MAPPER, 这样函数会根据给定的波形格式选择合适的设备}
```

```
  lpFormat: PWaveFormatEx; {TWaveFormat 结构的指针; TWaveFormat 包含要申请的波形格式}
  dwCallback: DWORD        {回调函数地址或窗口句柄; 若不使用回调机制, 设为nil}
  dwInstance: DWORD        {给回调函数的实例数据; 不用于窗口}
  dwFlags: DWORD           {打开选项}
): MMRESULT;               {成功返回 0; 可能的错误值见下:}

MMSYSERR_BADDEVICEID = 2;  {设备 ID 超界}
MMSYSERR_ALLOCATED   = 4;  {指定的资源已被分配}
MMSYSERR_NODRIVER    = 6;  {没有安装驱动程序}
MMSYSERR_NOMEM       = 7;  {不能分配或锁定内存}
WAVERR_BADFORMAT     = 32; {设备不支持请求的波形格式}

//TWaveFormatEx 结构:
TWaveFormatEx = packed record
  wFormatTag: Word;       {指定格式类型; 默认 WAVE_FORMAT_PCM = 1;}
  nChannels: Word;        {指出波形数据的通道数; 单声道为 1, 立体声为 2}
  nSamplesPerSec: DWORD;  {指定样本速率(每秒的样本数)}
  nAvgBytesPerSec: DWORD; {指定数据传输的平均速率(每秒的字节数)}
  nBlockAlign: Word;      {指定块对齐(单位字节), 块对齐是数据的最小单位}
  wBitsPerSample: Word;   {采样大小(字节)}
  cbSize: Word;           {附加信息大小; PCM 格式没这个字段}
end;
{16 位立体声 PCM 的块对齐是 4 字节(每个样本 2 字节, 2 个通道)}

//打开选项 dwFlags 的可选值:
WAVE_FORMAT_QUERY = $0001;     {只是判断设备是否支持给定的格式, 并不打开}
WAVE_ALLOWSYNC    = $0002;     {当是同步设备时必须指定}
CALLBACK_WINDOW   = $00010000; {当 dwCallback 是窗口句柄时指定}
CALLBACK_FUNCTION = $00030000; {当 dwCallback 是函数指针时指定}

//如果选择窗口接受回调信息, 可能会发送到窗口的消息有:
MM_WOM_OPEN  = $3BB;
```

```
MM_WOM_CLOSE = $3BC;
MM_WOM_DONE  = $3BD;

//如果选择函数接受回调信息, 可能会发送给函数的消息有:
WOM_OPEN  = MM_WOM_OPEN;
WOM_CLOSE = MM_WOM_CLOSE;
WOM_DONE  = MM_WOM_DONE;
```

```
//举例:
```

WinAPI: waveOutMessage - 向波形输出设备发送一条消息

```
//声明:
waveOutMessage(
  hWaveOut: HWAVEOUT; {设备句柄}
  uMessage: UINT;     {消息}
  dw1: DWORD          {消息参数}
  dw2: DWORD          {消息参数}
): Longint;           {将由设备给返回值}
```

```
//举例:
```

WinAPI: waveOutGetVolume - 获取输出设备当前的音量设置

提示:

参数 lpdwVolume 的两低位字节存放左声道音量, 两高位字节存放右声道音量;

$FFFF、$0000 分别表示最大与最小音量;

如不支持立体声, 两低位字节存放单声道音量.

```
//声明:
waveOutGetVolume(
  hwo: HWAVEOUT;     {设备句柄}
  lpdwVolume: PDWORD {存放音量值的变量的指针}
): MMRESULT;         {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE  = 5;  {设备句柄无效}
MMSYSERR_NODRIVER     = 6;  {没有安装驱动程序}
MMSYSERR_NOTSUPPORTED = 8;  {设备不支持}
```

```
//举例:
```

WinAPI: waveOutGetPosition - 获取输出设备当前的播放位置

```
//声明:
waveOutGetPosition(
  hWaveOut: HWAVEOUT; {设备句柄}
  lpInfo: PMMTime;    {TMMTime 结构的指针, 用于返回播放位置}
  uSize: UINT         {TMMTime 结构的大小, 以字节为单位}
): MMRESULT;          {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE  = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY   = 12; {设备已被另一线程使用}

//TMMTime 是 mmtime_tag 结构的重定义:
mmtime_tag = record
  case wType: UINT of
   TIME_MS:    (ms: DWORD);         {毫米}
   TIME_SAMPLES:(sample: DWORD);     {波形音频取样数}
   TIME_BYTES:  (cb: DWORD);         {波形音频字节数(字节偏移量)}
   TIME_TICKS:  (ticks: DWORD);      {TICK 数}
```

```
   TIME_SMPTE:(                    {动画及电视协会的 SMPTE 时间, 是个内嵌结构}
      hour: Byte;                  {时}
      min: Byte;                   {分}
      sec: Byte;                   {秒}
      frame: Byte;                 {帧}
      fps: Byte;                   {每秒帧数}
      dummy: Byte;                 {填充字节(为对齐而用)}
      pad: array[0..1] of Byte); {}
    TIME_MIDI : (songptrpos: DWORD); {MIDI 时间}
end;

//使用 TMMTime 结构前, 应先指定 TMMTime.wType :
TIME_MS      = $0001; {默认; 打开或复位时将回到此状态}
TIME_SAMPLES = $0002;
TIME_BYTES   = $0004;
TIME_SMPTE   = $0008;
TIME_MIDI    = $0010;
TIME_TICKS   = $0020;
```

```
//举例:
```

WinAPI: waveOutGetPlaybackRate - 获取输出设备当前的播放速度设置(默认速度值的倍数)

提示:

参数 lpdwRate 虽然指向的是 4 字节的正整数, 但表示的是个小数;

两个高位表示整数部分, 两个低位表示小数部分;

$8000 表示一半, $4000 表示四分之一;

譬如: $00010000 表示 1.0, 说明速度没有改变变; $000F8000, 表示 15.5 倍;

修改播放速度不会改变采样速度, 但肯定会改变播放时间.

```
//声明:
waveOutGetPlaybackRate(
  hWaveOut: HWAVEOUT; {设备句柄}
  lpdwRate: PDWORD    {存放速度值的变量的指针}
): MMRESULT;          {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE  = 5;  {设备句柄无效}
MMSYSERR_NOTSUPPORTED = 8;  {设备不支持}
MMSYSERR_HANDLEBUSY   = 12; {设备已被另一线程使用}
```

```
//举例:
```

WinAPI: waveOutGetPitch - 获取输出设备当前的音调设置(音高的倍数值)

提示:

参数 lpdwPitch 虽然指向的是 4 字节的正整数, 但表示的是个小数;

两个高位表示整数部分, 两个低位表示小数部分;

$8000 表示一半, $4000 表示四分之一;

譬如: $00010000 表示 1.0, 说明音高没变; $000F8000, 表示 15.5 倍;

修改音高不会改变播放速度、采样速度和播放时间, 但不是所有设备都支持.

```
//声明:
waveOutGetPitch(
  hWaveOut: HWAVEOUT; {设备句柄}
  lpdwPitch: PDWORD   {存放音高值的变量的指针}
): MMRESULT;          {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE  = 5;  {设备句柄无效}
MMSYSERR_NOTSUPPORTED = 8;  {设备不支持}
MMSYSERR_HANDLEBUSY   = 12; {设备已被另一线程使用}
```

```
//举例:
```

WinAPI: waveOutGetNumDevs - 获取波形输出设备的数目

```
//声明:
waveOutGetNumDevs: UINT; {无参数; 返回波形输出设备的数目}
```

```
//举例:
```

WinAPI: waveOutGetID - 获取输出设备 ID

```
//声明:
waveOutGetID(
  hWaveOut: HWAVEOUT; {设备句柄}
  lpuDeviceID: PUINT  {接受 ID 的变量的指针}
): MMRESULT;          {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}
```

```
//举例:
```

WinAPI: waveOutGetDevCaps - 查询输出设备的性能

```
//声明:
waveOutGetDevCaps(
  uDeviceID: UINT;        {输出设备 ID}
  lpCaps: PWaveOutCaps;   {TWaveOutCaps 结构的指针, 用于接受设备信息}
```

```
  uSize: UINT): MMRESULT; {TWaveOutCaps 结构大小}
): MMRESULT;              {成功返回 0; 可能的错误值见下:}

MMSYSERR_BADDEVICEID = 2; {设备 ID 超界}
MMSYSERR_NODRIVER    = 6; {没有安装驱动程序}

//TWaveOutCaps 是 tagWAVEOUTCAPSA 结构的重定义:
tagWAVEOUTCAPSA = record
  wMid: Word;                                  {制造商 ID}
  wPid: Word;                                  {产品 ID}
  vDriverVersion: MMVERSION;                      {版本号; 高字节是主版本号,
 低字节是次版本号}
  szPname: array[0..MAXPNAMELEN-1] of AnsiChar; {产品名称}
  dwFormats: DWORD;                             {支持的格式}
  wChannels: Word;                              {单声道(1)还是立体声(2)}
  dwSupport: DWORD;                             {其他功能}
end;

//dwFormats:
WAVE_INVALIDFORMAT = $00000000; {invalid format}
WAVE_FORMAT_1M08   = $00000001; {11.025 kHz, Mono,   8-bit }
WAVE_FORMAT_1S08   = $00000002; {11.025 kHz, Stereo, 8-bit }
WAVE_FORMAT_1M16   = $00000004; {11.025 kHz, Mono,   16-bit}
WAVE_FORMAT_1S16   = $00000008; {11.025 kHz, Stereo, 16-bit}
WAVE_FORMAT_2M08   = $00000010; {22.05  kHz, Mono,   8-bit }
WAVE_FORMAT_2S08   = $00000020; {22.05  kHz, Stereo, 8-bit }
WAVE_FORMAT_2M16   = $00000040; {22.05  kHz, Mono,   16-bit}
WAVE_FORMAT_2S16   = $00000080; {22.05  kHz, Stereo, 16-bit}
WAVE_FORMAT_4M08   = $00000100; {44.1   kHz, Mono,   8-bit }
WAVE_FORMAT_4S08   = $00000200; {44.1   kHz, Stereo, 8-bit }
WAVE_FORMAT_4M16   = $00000400; {44.1   kHz, Mono,   16-bit}
WAVE_FORMAT_4S16   = $00000800; {44.1   kHz, Stereo, 16-bit}

//dwSupport:
```

```
WAVECAPS_PITCH          = $0001; {支持音调控制}
WAVECAPS_PLAYBACKRATE   = $0002; {支持播放速度控制}
WAVECAPS_VOLUME         = $0004; {支持音量控制}
WAVECAPS_LRVOLUME       = $0008; {支持左右声道音量控制}
WAVECAPS_SYNC           = $0010; {}
WAVECAPS_SAMPLEACCURATE = $0020; {}
WAVECAPS_DIRECTSOUND    = $0040; {}
```

```
//举例:
```

WinAPI: waveOutClose - 关闭设备

提示: 若正在播放, 应先调用 waveOutReset 终止播放, 然后再关闭, 不然会失败.

```
//声明:
waveOutClose(
  hWaveOut: HWAVEOUT {设备句柄}
): MMRESULT;         {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}
WAVERR_STILLPLAYING  = 33; {缓冲区还在队列中}
```

```
//举例:
```

WinAPI: waveOutBreakLoop - 跳出循环

提示:

循环是由 saveOutWrite 传递的 TWaveHdr 结构的 dwLoop 和 dwFlags 控制的;

dwFlags 的 WHDR_BEGINLOOP、WHDR_ENDLOOP 标识循环的开始和结束数据块;
在同一数据块上循环，应同时指定这两个标志;
循环次数 dwLoops 应该在开始块上指定;
循环终止前，组成循环体的块一定要播放完;
当无播放内容或循环设定失败时，函数也能返回 0.

```
//声明:
waveOutBreakLoop(
  hWaveOut: HWAVEOUT {设备句柄}
): MMRESULT;         {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}


//TWaveHdr 是 wavehdr_tag 结构的重定义
wavehdr_tag = record
  lpData: PChar;          {指向波形数据缓冲区}
  dwBufferLength: DWORD;  {波形数据缓冲区的长度}
  dwBytesRecorded: DWORD; {若首部用于输入, 指出缓冲区中的数据量}
  dwUser: DWORD;          {指定用户的 32 位数据}
  dwFlags: DWORD;         {缓冲区标志}
  dwLoops: DWORD;         {循环播放次数, 仅用于输出缓冲区}
  lpNext: PWaveHdr;       {保留}
  reserved: DWORD;        {保留}
end;

//TWaveHdr 中的 dwFlags 的可选值:
WHDR_DONE      = $00000001; {设备已使用完缓冲区, 并返回给程序}
WHDR_PREPARED  = $00000002; {waveInPrepareHeader 或 waveOutPrepareH
eader 已将缓冲区准备好}
WHDR_BEGINLOOP = $00000004; {缓冲区是循环中的第一个缓冲区, 仅用于输出}
WHDR_ENDLOOP   = $00000008; {缓冲区是循环中的最后一个缓冲区, 仅用于输出}
WHDR_INQUEUE   = $00000010; {保留(给设备)}
```

//举例:

WinAPI: waveInUnprepareHeader - 清除由 waveInPrepareHeader 完成的准备

提示:

设备写满缓冲区返回给程序后，须调用此函数;

释放(GlobalFree)缓冲区前，须调用此函数;

取消一个尚未准备的缓冲区将无效，但函数返回 0

```
//声明:
waveInUnprepareHeader(
  hWaveIn: HWAVEIN;      {设备句柄}
  lpWaveInHdr: PWaveHdr; {TWaveHdr 结构的指针}
  uSize: UINT           {TWaveHdr 结构大小}
): MMRESULT;            {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}
WAVERR_STILLPLAYING  = 33; {缓冲区还在队列中}

//TWaveHdr 是 wavehdr_tag 结构的重定义
wavehdr_tag = record
  lpData: PChar;          {指向波形数据缓冲区}
  dwBufferLength: DWORD;  {波形数据缓冲区的长度}
  dwBytesRecorded: DWORD; {若首部用于输入, 指出缓冲区中的数据量}
  dwUser: DWORD;          {指定用户的 32 位数据}
  dwFlags: DWORD;         {缓冲区标志}
  dwLoops: DWORD;         {循环播放次数, 仅用于输出缓冲区}
  lpNext: PWaveHdr;       {保留}
  reserved: DWORD;        {保留}
```

```
end;

//TWaveHdr 中的 dwFlags 的可选值:
WHDR_DONE      = $00000001; {设备已使用完缓冲区, 并返回给程序}
WHDR_PREPARED  = $00000002; {waveInPrepareHeader 或 waveOutPrepareH
eader 已将缓冲区准备好}
WHDR_BEGINLOOP = $00000004; {缓冲区是循环中的第一个缓冲区, 仅用于输出}
WHDR_ENDLOOP   = $00000008; {缓冲区是循环中的最后一个缓冲区, 仅用于输出}
WHDR_INQUEUE   = $00000010; { reserved for driver }
```

```
//举例:
```

WinAPI: waveInStop - 停止输入

提示: 如果未启动则调用无效, 但也返回 0; 缓冲区会被返回, TWaveHdr 结构中的 dwBytesRecorded 将包含返回的实际数据的长度.

```
//声明:
waveInStop(
  hWaveIn: HWAVEIN {设备句柄}
): MMRESULT;       {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}
```

```
//举例:
```

WinAPI: waveInStart - 启动输入

```
//声明:
waveInStart(
  hWaveIn: HWAVEIN {设备句柄}
): MMRESULT;       {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}
```

```
//举例:
```

WinAPI: waveInReset - 重置输入

提示:

函数会终止输入，位置清 0；放弃未处理的缓冲区并返回给程序；

TWaveHdr 结构中的 dwBytesRecorded 将包含实际数据的长度.

```
//声明:
waveInReset(
  hWaveIn: HWAVEIN {设备句柄}
): MMRESULT;       {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}
```

```
//举例:
```

WinAPI: waveInPrepareHeader - 为波形输入准备一个缓冲区

提示：必须调用 GlobalAlloc 给 TWaveHdr 和其中的 lpData 指向的缓冲区分配内存(使用 GMEM_MOVEABLE、GMEM_SHARE)，并用 GlobalLock 锁定.

```
//声明:
waveInPrepareHeader(
  hWaveIn: HWAVEIN;      {设备句柄}
  lpWaveInHdr: PWaveHdr; {TWaveHdr 结构的指针}
  uSize: UINT           {TWaveHdr 结构大小}
): MMRESULT;            {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_NOMEM      = 7;  {不能分配或锁定内存}
MMSYSERR_HANDLEBUSY  = 12; {其他线程正在使用该设备}

//TWaveHdr 是 wavehdr_tag 结构的重定义
wavehdr_tag = record
  lpData: PChar;         {指向波形数据缓冲区}
  dwBufferLength: DWORD;  {波形数据缓冲区的长度}
  dwBytesRecorded: DWORD; {若 TWaveHdr 用于输入, 指出缓冲区中的数据量}
  dwUser: DWORD;         {指定用户的 32 位数据}
  dwFlags: DWORD;        {缓冲区标志}
  dwLoops: DWORD;        {循环播放次数, 仅用于输出缓冲区}
  lpNext: PWaveHdr;      {保留}
  reserved: DWORD;       {保留}
end;

//TWaveHdr 中的 dwFlags 的可选值:
WHDR_DONE     = $00000001; {设备已使用完缓冲区, 并返回给程序}
WHDR_PREPARED  = $00000002; {waveInPrepareHeader 或 waveOutPrepareH
eader 已将缓冲区准备好}
WHDR_BEGINLOOP = $00000004; {缓冲区是循环中的第一个缓冲区, 仅用于输出}
WHDR_ENDLOOP   = $00000008; {缓冲区是循环中的最后一个缓冲区, 仅用于输出}
WHDR_INQUEUE   = $00000010; { reserved for driver }
```

```
//举例:
```

WinAPI: waveInOpen - 打开波形输入设备

提示: 因为其中的回调函数是在中断时间内访问的, 必须在 DLL 中; 要访问的数据都必须是在固定的数据段中; 除了

PostMessage

timeGetSystemTime

timeGetTime

timeSetEvent

timeKillEvent

midiOutShortMsg

midiOutLongMsg

OutputDebugString 外, 也不能有其他系统调用.

```
//声明:
waveInOpen(
  lphWaveIn: PHWAVEIN;      {用于返回设备句柄的指针; 如果 dwFlags=WAVE_FORMAT_QUERY, 这里应是 nil}
  uDeviceID: UINT;          {设备 ID; 可以指定为: WAVE_MAPPER, 这样函数会根据给定的波形格式选择合适的设备}
  lpFormatEx: PWaveFormatEx; {TWaveFormat 结构的指针; TWaveFormat 包含要申请的波形格式}
  dwCallback: DWORD         {回调函数地址或窗口句柄; 若不使用回调机制, 设为 nil}
  dwInstance: DWORD         {给回调函数的实例数据; 不用于窗口}
  dwFlags: DWORD            {打开选项}
): MMRESULT;                {成功返回 0; 可能的错误值见下:}

MMSYSERR_BADDEVICEID = 2;  {设备 ID 超界}
MMSYSERR_ALLOCATED   = 4;  {指定的资源已被分配}
MMSYSERR_NODRIVER    = 6;  {没有安装驱动程序}
```

```
MMSYSERR_NOMEM      = 7;  {不能分配或锁定内存}
WAVERR_BADFORMAT    = 32; {设备不支持请求的波形格式}

//TWaveFormatEx 结构:
TWaveFormatEx = packed record
  wFormatTag: Word;      {指定格式类型; 默认 WAVE_FORMAT_PCM = 1;}
  nChannels: Word;       {指出波形数据的声道数; 单声道为 1, 立体声为 2}
  nSamplesPerSec: DWORD;  {指定采样频率(每秒的样本数)}
  nAvgBytesPerSec: DWORD; {指定数据传输的传输速率(每秒的字节数)}
  nBlockAlign: Word;     {指定块对齐(每个样本的字节数), 块对齐是数据的最
小单位}
  wBitsPerSample: Word;   {采样大小(字节), 每个样本的量化位数}
  cbSize: Word;          {附加信息的字节大小}
end;
{16 位立体声 PCM 的块对齐是 4 字节(每个样本 2 字节, 2 个通道)}

//打开选项 dwFlags 的可选值:
WAVE_FORMAT_QUERY = $0001;     {只是判断设备是否支持给定的格式, 并不打开}
WAVE_ALLOWSYNC    = $0002;     {当是同步设备时必须指定}
CALLBACK_WINDOW   = $00010000; {当 dwCallback 是窗口句柄时指定}
CALLBACK_FUNCTION = $00030000; {当 dwCallback 是函数指针时指定}

//如果选择窗口接受回调信息, 可能会发送到窗口的消息有:
MM_WIM_OPEN  = $3BE;
MM_WIM_CLOSE = $3BF;
MM_WIM_DATA  = $3C0;

//如果选择函数接受回调信息, 可能会发送给函数的消息有:
WIM_OPEN  = MM_WIM_OPEN;
WIM_CLOSE = MM_WIM_CLOSE;
WIM_DATA  = MM_WIM_DATA;
```

```
//举例:
```

WinAPI: waveInMessage - 向波形输入设备发送一条消息

```
//声明:
waveInMessage(
  hWaveIn: HWAVEIN; {设备句柄}
  uMessage: UINT;   {消息}
  dw1: DWORD        {消息参数}
  dw2: DWORD        {消息参数}
): MMRESULT;        {将由设备给返回值}
```

```
//举例:
```

WinAPI: waveInGetPosition - 获取当前输入设备的输入位置

```
//声明:
waveInGetPosition(
  hWaveIn: HWAVEIN; {设备句柄}
  lpInfo: PMMTime;  {TMMTime 结构的指针}
  uSize: UINT       {TMMTime 结构大小}
): MMRESULT;        {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}

//TMMTime 是 mmtime_tag 结构的重定义:
mmtime_tag = record
  case wType: UINT of
   TIME_MS:     (ms: DWORD);         {毫米}
   TIME_SAMPLES:(sample: DWORD);     {波形音频取样数}
   TIME_BYTES:  (cb: DWORD);         {波形音频字节数(字节偏移量)}
   TIME_TICKS:  (ticks: DWORD);      {TICK 数}
   TIME_SMPTE:(                     {动画及电视协会的 SMPTE 时间, 是个内嵌
结构}
     hour: Byte;              {时}
```

```
      min: Byte;                {分}
      sec: Byte;                {秒}
      frame: Byte;              {帧}
      fps: Byte;                {每秒帧数}
      dummy: Byte;              {填充字节(为对齐而用)}
      pad: array[0..1] of Byte); {}
    TIME_MIDI : (songptrpos: DWORD); {MIDI 时间}
end;

//使用 TMMTime 结构前, 应先指定 TMMTime.wType :
TIME_MS      = $0001; {默认; 打开或复位时将回到此状态}
TIME_SAMPLES = $0002;
TIME_BYTES   = $0004;
TIME_SMPTE   = $0008;
TIME_MIDI    = $0010;
TIME_TICKS   = $0020;
```

```
//举例:
```

WinAPI: waveInGetNumDevs - 获取波形输入设备的数目

```
//声明:
waveInGetNumDevs: UINT; {无参数; 返回波形输入设备的数目}
```

```
//举例:
```

WinAPI: waveInGetID - 获取输入设备 ID

```
//声明:
```

```
waveInGetID(
  hWaveIn: HWAVEIN;  {获取输入设备句柄}
  lpuDeviceID: PUINT {接受 ID 的变量的指针}
): MMRESULT;         {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}
```

```
//举例:
```

WinAPI: waveInGetErrorText - 根据错误号得到错误描述

提示: 错误文本的长度一般不超过 MAXERRORLENGTH = 128; 如果缓冲区太小, 文本会被截断.

```
//声明:
waveInGetErrorText(
  mmrError: MMRESULT; {错误号}
  lpText: PChar;      {缓冲区}
  uSize: UINT         {缓冲区大小}
): MMRESULT;          {成功返回 0; 失败再返回错误号, 可能的错误是:}

MMSYSERR_BADERRNUM = 9; {错误号超界}
```

```
//举例:
```

WinAPI: waveInGetDevCaps - 查询输入设备的性能

```
//声明:
```

```
waveInGetDevCaps(
  hwo: HWAVEOUT;       {输入设备 ID; HWAVEIN ?}
  lpCaps: PWaveInCaps; {TWaveInCaps 结构的指针, 用于接受设备信息}
  uSize: UINT          {TWaveInCaps 结构大小}
): MMRESULT;           {成功返回 0; 可能的错误值见下:}

MMSYSERR_BADDEVICEID = 2; {设备 ID 超界}
MMSYSERR_NODRIVER    = 6; {没有安装驱动程序}

//TWaveInCaps 是 tagWAVEINCAPSA 结构的重定义:
tagWAVEINCAPSA = record
  wMid: Word;                                  {制造商 ID}
  wPid: Word;                                  {产品 ID}
  vDriverVersion: MMVERSION;                      {版本号; 高字节是主版本号,
 低字节是次版本号}
  szPname: array[0..MAXPNAMELEN-1] of AnsiChar; {产品名称}
  dwFormats: DWORD;                             {支持的格式}
  wChannels: Word;                              {单声道(1)还是立体声(2)}
  wReserved1: Word;                             { structure packing }
end;

//dwFormats:
WAVE_INVALIDFORMAT = $00000000; {invalid format}
WAVE_FORMAT_1M08   = $00000001; {11.025 kHz, Mono,   8-bit }
WAVE_FORMAT_1S08   = $00000002; {11.025 kHz, Stereo, 8-bit }
WAVE_FORMAT_1M16   = $00000004; {11.025 kHz, Mono,   16-bit}
WAVE_FORMAT_1S16   = $00000008; {11.025 kHz, Stereo, 16-bit}
WAVE_FORMAT_2M08   = $00000010; {22.05  kHz, Mono,   8-bit }
WAVE_FORMAT_2S08   = $00000020; {22.05  kHz, Stereo, 8-bit }
WAVE_FORMAT_2M16   = $00000040; {22.05  kHz, Mono,   16-bit}
WAVE_FORMAT_2S16   = $00000080; {22.05  kHz, Stereo, 16-bit}
WAVE_FORMAT_4M08   = $00000100; {44.1   kHz, Mono,   8-bit }
WAVE_FORMAT_4S08   = $00000200; {44.1   kHz, Stereo, 8-bit }
WAVE_FORMAT_4M16   = $00000400; {44.1   kHz, Mono,   16-bit}
```

```
WAVE_FORMAT_4S16   = $00000800; {44.1   kHz, Stereo, 16-bit}
```

```
//举例:
```

WinAPI: waveInClose - 关闭指定的波形输入设备

提示: 若 waveInAddBuffer 送出的缓冲区未返回则失败; 可用 waveInReset 放弃所有未用完的缓冲区.

```
//声明:
waveInClose(
  hWaveIn: HWAVEIN {设备句柄; 函数若成功返回, 句柄则不再有效}
): MMRESULT;       {成功返回 0; 可能的错误值见下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
WAVERR_STILLPLAYING  = 33; {缓冲区还在队列中}
```

```
//举例:
```

WinAPI: waveInAddBuffer - 向波形输入设备发送一个输入缓冲区

提示:

缓冲区写满后送回应用程序.

在缓冲区给 waveInAddBuffer 前, 先要调用 waveInPrepareHeader 准备; 还要调用 GlobalAlloc 给 TWaveHdr 和其中 lpData 指向的缓冲区分配内存(使用 GMEM_MOVEABLE、GMEM_SHARE), 并用 GlobalLock 锁定.

```
//声明:
waveInAddBuffer(
```

```
  hWaveIn: HWAVEIN;      {波形输入设备句柄}
  lpWaveInHdr: PWaveHdr; {TWaveHdr 结构的指针}
  uSize: UINT            {TWaveHdr 结构大小}
): MMRESULT;             {成功返回 0; 可能的错误值如下:}

MMSYSERR_INVALHANDLE = 5;  {设备句柄无效}
WAVERR_UNPREPARED    = 34; {没准备好缓冲区}
MMSYSERR_HANDLEBUSY  = 12; {设备已被另一线程使用}

//TWaveHdr 是 wavehdr_tag 结构的重定义
wavehdr_tag = record
  lpData: PChar;         {指向波形数据缓冲区}
  dwBufferLength: DWORD;  {波形数据缓冲区的长度}
  dwBytesRecorded: DWORD; {若首部用于输入, 指出缓冲区中的数据量}
  dwUser: DWORD;         {指定用户的 32 位数据}
  dwFlags: DWORD;        {缓冲区标志}
  dwLoops: DWORD;        {循环播放次数, 仅用于输出缓冲区}
  lpNext: PWaveHdr;      {保留}
  reserved: DWORD;       {保留}
end;

//dwFlags 的可选值:
WHDR_DONE      = $00000001; {设备已使用完缓冲区, 并返回给程序}
WHDR_PREPARED  = $00000002; {waveInPrepareHeader 或 waveOutPrepareH
eader 已将缓冲区准备好}
WHDR_BEGINLOOP = $00000004; {缓冲区是循环中的第一个缓冲区, 仅用于输出}
WHDR_ENDLOOP   = $00000008; {缓冲区是循环中的最后一个缓冲区, 仅用于输出}
WHDR_INQUEUE   = $00000010; { reserved for driver }
```

```
//举例:
```

合并两个 Wav 文件的函数

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Control
s, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

{合并两个 Wav 文件的函数}
function ConWavFile(AWavFile1, AWavFile2, ANewFile: string): Boole
an;
type
  TWavFormat = packed record
    ChunkID: array[0..3] of AnsiChar;     {'RIFF'}
    ChunkSize: Longword;                {size-8}
    Format: array[0..3] of AnsiChar;      {'WAVE'}
    SubChunk1ID: array[0..3] of AnsiChar; {'fmt '}
    SubChunk1Size: Longword;             {hex10}
    AudioFormat: Word;                  {hex 01}
```

```
    NumOfChannels: Word;               {1 mono, 2 stereo}
    SampleRate: Longword;              {number of samples/sec}
    ByteRate: Longword;                {samplerate* num of channels
*bytes per (mono) sample}
    BytesperSample: Word;              {size of (mono) sample}
    BitsPerSample: Word;               {BytesperSample *8}
    SubChunk2ID: array[0..3] of AnsiChar; {'data'}
    SubChunk2Size: Longword;           {number of data bytes}
  end;
var
  vWavFormat1: TWavFormat;
  vWavFormat2: TWavFormat;
  vFileHandle1: THandle;
  vFileHandle2: THandle;
  vFileStream1: TFileStream;
  vFileStream2: TFileStream;
  vChunkSize1, vChunkSize2: Integer;
begin
  Result := False;
  if not FileExists(AWavFile1) then Exit;
  if not FileExists(AWavFile2) then Exit;

  vFileHandle1 := _lopen(PAnsiChar(AnsiString(AWavFile1)), OF_READ
 or OF_SHARE_DENY_NONE);
  vFileHandle2 := _lopen(PAnsiChar(AnsiString(AWavFile2)), OF_READ
 or OF_SHARE_DENY_NONE);

  if (Integer(vFileHandle1) <= 0) or (Integer(vFileHandle2) <= 0) t
hen
  begin
    _lclose(vFileHandle1);
    _lclose(vFileHandle2);
    Exit;
  end;
```

```
  vFileStream1 := TFileStream.Create(vFileHandle1);
  vFileStream2 := TFileStream.Create(vFileHandle2);
  try
    if vFileStream1.Read(vWavFormat1, SizeOf(TWavFormat)) <> SizeOf
(TWavFormat) then Exit;
    if vFileStream2.Read(vWavFormat2, SizeOf(TWavFormat)) <> SizeOf
(TWavFormat) then Exit;
    if vWavFormat1.ChunkID <> 'RIFF' then Exit;
    if vWavFormat1.SubChunk2ID <> 'data' then Exit;
    vChunkSize1 := vWavFormat1.SubChunk2Size;
    vChunkSize2 := vWavFormat2.SubChunk2Size;
    vWavFormat1.ChunkSize := 0;
    vWavFormat1.SubChunk2Size := 0;
    vWavFormat2.ChunkSize := 0;
    vWavFormat2.SubChunk2Size := 0;
    if not CompareMem(@vWavFormat1, @vWavFormat2, SizeOf(TWavForma
t)) then Exit; {格式不同}
    with TMemoryStream.Create do try
      vWavFormat1.ChunkSize := vChunkSize1 + vChunkSize2 + SizeOf(v
WavFormat1) - 8;
      vWavFormat1.SubChunk2Size := vChunkSize1 + vChunkSize2;
      Write(vWavFormat1, SizeOf(TWavFormat));
      CopyFrom(vFileStream1, vChunkSize1);
      CopyFrom(vFileStream2, vChunkSize2);
      try
        SaveToFile(ANewFile);
      except
        Exit;
      end;
    finally
      Free;
    end;
  finally
```

```
    vFileStream1.Free;
    vFileStream2.Free;
  end;
  Result := True;
end; { ConWavFile End}


{测试}
procedure TForm1.Button1Click(Sender: TObject);
var
  Wav1,Wav2,WavDest: string;
begin
  Wav1 := 'c:\temp\1.wav';
  Wav2 := 'c:\temp\2.wav';
  WavDest := 'c:\temp\12.wav';
  if ConWavFile(Wav1, Wav2, WavDest) then
    ShowMessageFmt('''%s'' 和 ''%s'' 已成功合并到 ''%s''', [Wav1,Wav2,
WavDest]);
end;

end.
```
