Dr. Edgar Huckert
12-2015, V1.2
Mail: edgar.huckert@huckert.com

Playing synthesized chords under Windows

Overview

Playing sounds under Windows is surprisingly complicated if you want to implement musical applications that go beyond the playing of simple files. My main interest was to a build music player that could play **polyphonic music** where several sounds (called „tones") can be combined into chords or independent voices. Note that is also possible to hear a chord when you use a pre-mixed WAVE file, i.e. a WAVE file that contains two or more tones sounding at the same time (a chord recorded from a piano or a group of instruments). But thisassumes external pre-mixing – which is not our goal.

Playing **multiple sounds** at the same time can also be used to produce acoustic effects. The simplest effect is the **tremolo** which can easily be produced by playing a normal sound (sinus, saw tooth etc.) combined with a slow sine wave (1 – 3 Hz).

Some music instruments use a similar process to produce complex sounds. The most famous example is the classic (electric, not electronic) **Hammond organ**, where sinus-like basic sounds are mixed, modulated, amplified and sent to the speakers. The classic synthesizers (Moog type) also use this approach whereas more recent synthesizers use completely different procedures – they are rather sample players than synthesizers.

The Microsoft APIs offer three types of sound functions

1. very simple but unflexible methods (Windows API)
2. low level, flexible methods (Windows API)
3. methods used in the context of game programming (DirectSound API) - we don't discuss that here, see [Scherfgen, p 418 ff]

To make things simple we assume here that sounds are stored in WAVE files. WAVE files are a special case of RIFF files. The class *WaveFile* shown in the second sample can be used to decode WAVE files (see member function *readAttributes*()). You can also use the simpler sample file *wavedecode.c* in the companion ZIP file. We don't explain WAVE files here in detail. You should however know that WAVE file have three parts:

1. a fixed header showing the tags „RIFF" and „WAVE"
2. the essential acoustic attributes (after the tag „fmt „)
3. one or more data chunks, one must be identified by the tag „data"

In a real music player based on our technique sounds should be played from memory instead of files – but this is not our main concern here.

As usual I use here C++ as programming language. My favorite compiler has always been the C++ compiler from Digital Mars (in the earlier days called Symantec C++). I see however no problems in using other compilers like g++ (GNU) or Visual C++ (Microsoft). You must link in the library *winmm.lib*.

Other programming languages like Basic or C# can also be used with some modifications. I have also checked some of the usual C++ based general class libraries like *wxWidgets* (I prefer that for building cross platform applications): none of them has a complex sound API as required here.

The sample programs are **console mode** programs to be run in a command window („black window"). Implementing them as graphical Windows programs would have produced larger source code without adding anything essential for our problem.

Using simple API methods

The most simple sound player methods are *sndPlaySound*() or the newer *PlaySound*() method. These methods accept file names as input (or memory handles etc. - we don't discuss that here). The C++ source code can be found under „sample 1" below. The main advantages for this approach are:

- no need to know the internal structure of the WAVE files
- no need to know anything about sample rates, number of channels, encodings etc.
- no need to open or close the output devices

The methods *sndPlaySound*() and *PlaySound*() allow asynchronous mode – this is essential for us. „Asynchronous" means: the called method returns immediately back to the caller without waiting for the competion of the sound. The documentation does not mention events nor messages that are sent to the application when essential changes („end-of-play", „buffer empty" etc.) in the player occur. You can stop the player at any time by calling *sndPlaySound()* with a NULL argument but this must be based on our own timing.

There are some several drawbacks if you want to implement real musical applications like players for Midi base on own sample files (without using the MIDI player from Windows). In my tests the major drawback was the **inability to play synthesized chords**, i.e. chords that are synthesized from simple (monophonic) sound files. The normal C major chord (triad c-e-g) should be based on three simple sound files that are played simultaneously.

Even with a rather sophisticated approach – using a thread for each component of the chord and using the *asynchronous* flag – the simple API calls did not work. There are no details in the Microsoft documentation for the simple API

calls. I guess however that the WAVE output device is opened and closed each time you call the API methods so that subsequent calls destroy previous sounds.

The simple API functions offer some additional features – like playing from memory. These features are however of no interest for our problem. What is really missing is the **callback mechanism** that we use in the next chapter.
The code using the simple API calls is in a source file called *waveClass1.cpp*. In order to use the simple API methods call the demo program *waveClass1* on the command like like this (using two input WAVE files):

```
waveClass1 -i g_1.wav,c_1.wav
```

If you want to compile and link the code you will find instructions in the comments of the source code. But be warned be warned: even if this sample suggests that you can play multiple files at the same time – it won't work. I found no solution for this problem! It would be nice if some reader could inform me if he has a solution by using the simple API functions.

Using more sophisticated API methods

Threads, events, callback functions

We have to come down to the more complex low level API functions to solve our problem: playing chords, i.e. multiple tones at the same time. The Windows „Wave Form Audio Functions" contain four calls used in our sample program:

- waveOutOpen()
- waveOutPrepareHeader()
- waveOutWrite()
- waveOutClose()

Note that there are more methods in the API - but these are the essential ones. From a conceptual point of view our problem is solved by using three rather complex mechanisms:

1. a callback procedure for the API calls
2. threads and associated worker routines
3. events indicating the end of the threads

A **callback procedure** specified in *WaveOutOpen()* is necessary to inform us if the player has reached certain states. We use the API message *MM_WOM_DONE* here to detect if the play for a single tone is over. All other messages are ignored.

You could also specify a **callback window** (not applicable here as we use a

command line program) or or an **callback event** (see *waveClass4.cpp* below) to obtain similar results. The callback mechanism indicates clearly the **asynchronous** nature of the main API call *waveOutWrite()*: it returns immediately after being called so that you must wait somewhere for messages sent by the audio driver.

Threads and their associated **worker routine** are started for each tone (each wave file given on the command line). Each thread creates an instance of the *WaveFile* class, starts the player (this is the API call *waveOutWrite()*) and waits for the *MM_WOM_DONE* message saying that the play is over. Using a callback window instead of a callback function could avoid threads as Microsoft sends in this case the messages into the central Windows message loop – I didn't test that however.

A **termination event** ist sent by each thread to the main thread as soon as it terminates. The main application waits until the last thread has terminated – see the Windows API call *WaitForMultipleObjects*(). If we would't use an event per thread than the application would risk to be terminated before the longest WAVE file was played entirely.

You should not confuse these termination events with the events that can be specified in the *waveOutOpen()* API function (see the next chapter).

It is essential that the sound device is opened (see call *WaveOutOpen()*) for each thread (each WAVE file). The device should only be closed after the reception of message *MM_WOM_DONE.*

Threads, events, no callback functions

We mentioned above that you can also use Windows **events** (see API calls *CreateEvent(), SetEvent(), WaitFor...()*) to get informed about the state of the players (threads). In fact this reduces the complexity of the class waveClass3 (see sample 2), as we need no callback function and the evaluation of the messages in this function. The only drawback is the fact that we need now **two types of events**:

- thread termination events
- player state change events

Note however that this event mechanims is not as granular as the callback function mechanism. We get no event for state changes like transitions to the open or close state. But for our actual requirements this is enough. Except for the detection of the end of the playing process and for some setup details (see *WaveOutOpen()* and *WaveOutPrepareHeader()*) the code is very similar to the previous soluton.

The detection of the end of the play process has to be changed: as soon as a player state change event arrives we must evaluate the *dwFlags* member of

the *WAVEHDR* structure given in the *WaveOutPrepareHeader()* call. If we detect there the flag *WHDR_DONE* the we sent to our main process a termination event and we terminate the player thread. Look at the *while* loop at the end of the *play()* member function to find out how this is implemented. This *while* loop is also given below under the heading **sample 3.**

The complete code for this version of our player can be found in the companion ZIP file under the name **waveClass4.cpp**. This version is about 40 lines smaller than *waveClass3.cpp* as we need no callback function. If you accept the verdict „smaller is better" than this should be your solution.

Practical hints

You can easily produce WAVE files for testing by using the „generate/tone generator 1" menu in the free tool *Audacity*. Assume that you have produced three WAVE files for a1=440 Hz, for g1 and e1. You can produce then a C major chord (triad) by issueing this command:

```
waveClass3 -i c_1.wav,e_1.wav,g_1,wav
```

The flag „-v" on the command line may be given to output some details on the file structure and the playing process.

The WAVE files may have different sizes. I assume that the other parameters (sampling rate, number of channels etc.) must be the same as the parameters are transferred to the sound system as soon as the first file is opend.

sample 1: using the simple API

```
// module waveClass1
// Copyright Dr.E.Huckert 07-2011
// Spielt eine Liste von Wave Dateien ab
// Only for Windows (MultiMedia Library winmm.lib)
//
// This version uses the simple API call sndPlayFile() in synchronous mode
// No threads or events are used!
//
// Compile with:
//  dmc -mn -WA waveClass1.cpp winmm.lib

#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <mmsystem.h>

class      WaveFile;
static int logging = 0;

// ------------------------------------------------
// A class to evaluate and play WAVE files. Can
// be used for simple API calls and more sophisticated calls
// class written by EH 01-2015
// modified/enhanced  11-2015
class WaveFile
```

```
{
  public:
    int           no_channels;
    int           bit_samp;
    int           bytes_samp;
    unsigned long sample_rate;
    unsigned long bytes_sec;
    unsigned long length;
    char          fileName[256];
    unsigned char *pData;
    //
  public:
    WaveFile(const char *file_name);  // constructor
    ~WaveFile();
    int readAttributes(int dataFlag); // checks also if the file exists
    int playFile();                   // uses sndPlaySound() to play
};

// ------------------------------------------
// constructor with agiven file name(WAV file)
WaveFile::WaveFile(const char *file_name)
{
  if (file_name == NULL)
    ::strcpy(this->fileName, "");
  else
    ::strcpy(this->fileName, file_name);
  this->no_channels  = 1;
  this->bytes_samp   = 0;
  this->bit_samp     = 0;
  this->length       = 0L;
  this->bytes_sec    = 0L;
  this->sample_rate  = 0L;
  this->pData        = NULL;
}

// ------------------------------------------------
// Destructor
WaveFile::~WaveFile()
{
  if (this->pData != NULL)
    delete this->pData;
}

// ------------------------------------------------
// Open the file, evaluate the attributes.
// Read and keep the data of flag dataFlag is set
int WaveFile::readAttributes(int dataFlag)
{
  int   ret = 0;
  int   cRet = 0;
  FILE  *stream = NULL;
  char  buff[5];
  int   state = 0;
  long  offset;
  //
  // Very important here: "rb" with "b"=binary!!!!!!
  stream = ::fopen(this->fileName, "rb");
  if (stream == NULL)
  {
    ret = -1;
    goto zurueck;
  }
  // the  file is here open
  buff[4]='\0';
  // read the first 4 bytes
  ::fread((void *)buff,1,4,stream);
  // the first four bytes should be 'RIFF'
  if (::strcmp((char *)buff,"RIFF")== 0)
  {
      state = 1;
      // read byte 8,9,10 and 11
```

```
::fseek(stream, 4, SEEK_CUR);
::fread((void *)buff,1,4,stream);
// this should read "WAVE"
if (::strcmp((char *)buff,"WAVE")== 0)
{
    state = 2;
    // read byte 12,13,14,15
    ::fread((void *)buff,1,4,stream);
    // this should read "fmt "
    if (::strcmp((char *)buff,"fmt ")== 0)
    {
      // decode the essential attributes
      ::fseek(stream, 20, SEEK_CUR);
      //  read byte 36,37,38,39, =Pos. 0x24 ff
      ::fread((void *)buff, 1, 4, stream);
      if (strcmp((char *)buff,"data")== 0)
      {
        // Now we know it is a wav file, rewind the stream
        ::rewind(stream);
        // is it no_channels or stereo ?
        ::fseek(stream, 22, SEEK_CUR);
        ::fread((void *)buff, 1, 2,stream);
        no_channels = buff[0];
        // read the sample rate and other parameters
        ::fread((void *)&(this->sample_rate), 1, 4, stream);
        ::fread((void *)&(this->bytes_sec), 1, 4, stream);
        ::fread((void *)&(this->bytes_samp), 1 ,2, stream);
        ::fread((void *)&(this->bit_samp), 1, 2, stream);
        ::fseek(stream, 4, SEEK_CUR);
        ::fread((void *)&(this->length), 1, 4, stream);
        // = 0L;
        if (dataFlag)
        {
          // read and keep the data (samples)
          cRet = ::fseek(stream, 0x2C, SEEK_SET);
          if (::logging)
             ::printf("WaveFile::readAttributes() fseek=%d\n", cRet);
          offset = ::ftell(stream);
          if (::logging)
             ::printf("WaveFile::readAttributes() ftell=%ld\n", offset);
          this->pData = new unsigned char[this->length];
          long ll = this->length;
          long nbRead = 0L;
          int nb;
          unsigned char *ptr = this->pData;
          if (::logging)
           ::printf("WaveFile::readAttributes() ll=%ld\n", ll);
          //
          // Using a loop here seems to be unnecessary when using fread
          // A loop is however necessary when using read() instead
          while (ll > 0L)
          {
            nb = 4096;
            if (nb < ll) nb = ll;
            nb = ::fread((void *)ptr, 1, nb, stream);
            if (::logging)
             ::printf("WaveFile::readAttributes() nb=%d\n", nb);
            ptr += nb;
            if (nb <= 0)
              break;
            nbRead += nb;
            ll -= nb;
            if (ferror(stream))
            {
              if (::logging)
               ::printf("WaveFile::readAttributes() ferror\n");
              break;
            }
            if (feof(stream))
            {
              if (::logging)
```

```
                    ::printf("WaveFile::readAttributes() feof\n");
                  break;
                }
              }   // end while (ll > 0L)
              if (::logging)
                ::printf("WaveFile::readAttributes() data read=%ld\n", nbRead);
            }   // end if (dataFlag)
          }
        }
      }
  }
  //
  if (state != 2)
  {
    ret = -2;
    goto zurueck;
  }
  if (::logging)
  {
    ::printf("WaveFile::readAttributes() ret=%d\n", ret);
    ::printf("WaveFile::readAttributes() no_channels=%d\n", this->no_channels);
    ::printf("WaveFile::readAttributes() length=%ld\n", this->length);
    ::printf("WaveFile::readAttributes() sample_rate=%ld\n", this->sample_rate);
    ::printf("WaveFile::readAttributes() bytes_samp=%d\n", this->bytes_samp);
    ::printf("WaveFile::readAttributes() bytes_sec=%ld\n", this->bytes_sec);
  }
  //
  zurueck:
  if (stream != NULL)
    ::fclose(stream);
  if (::logging)
    ::printf("WaveFile::readAttributes() ret=%d\n", ret);
  return ret;
}   // end WaveFile::readAttributes()

// ------------------------------------------------------
// Eine WAVE-Datei uber sndPlaySound() oeffnen und abspielen
// Warning: cannot be used to play chords, i.e. multiple wave file simultaneously!!!!!
int WaveFile::playFile()
{
  int ret = 0;
  int cRet;
  //
  if (::logging)
    ::printf("WaveFile::playFile() file=%s\n", this->fileName);
  if (::strlen(this->fileName) == 0)
  {
    ret = -1;
    goto zurueck;
  }
  //
  // Muss nicht sein!
  cRet = this->readAttributes(0);
  //
  // we use here the simplest play function in the windows API: sndPlaySound()
  // EH: must be mode=SND_SYNC - if not only the first buffer is played and
  //        the rest is skipped as the application terminates!
  //       ret = ::sndPlaySound(filename, SND_FILENAME | SND_ASYNC);
  cRet = ::sndPlaySound(this->fileName, SND_FILENAME |
                                        SND_SYNC);
  if (cRet < 0)
  {
    ret = -2;
    goto zurueck;
  }
  //
  zurueck:
  if (::logging)
    ::printf("WaveFile::playFile() ret=%d\n", ret);
  return ret;
}   // end playFile()
```

```
// ------------------------------------------------------
int getNextFn(char *strFn, char *actFn, int startPos)
{
  int pos  = startPos;
  int i    = 0;
  actFn[0] = 0;
  while (strFn[pos] != 0)
  {
    if (strFn[pos] == ' ')
    {
      pos++;
      continue;
    }
    if (strFn[pos] == '"')
    {
      pos++;
      continue;
    }
    if (strFn[pos] == 0)
      break;
    if (strFn[pos] == ',')
    {
      pos++;
      break;
    }
    actFn[i++] = strFn[pos];
    actFn[i]   = 0;
    pos++;
  }
  if (::strlen(actFn) == 0)
    pos = -1;
  return pos;
}   // end getNextFn()

// ------------------------------------------------------
void usage()
{
  ::printf("usage: waveClass1 -i fileList [-v]\n");
}   // end usage()

// ------------------------------------------------------
int main(int argc, char *argv[])
{
  int      ret = 0;
  int      cRet;
  int      startPos = 0;
  char     waveFn[512] = "";
  char     actFn[256]  = "";
  WaveFile *pwf = NULL;
  //
  // the  filelistist a list of comma separated file names (WAVE files)
  for (int n=1; n < argc; n++)
  {
    if (::strcmp(argv[n], "-i") == 0)
      ::strcpy(waveFn, argv[n + 1]);
    if (::strcmp(argv[n], "-v") == 0)
      ::logging = 1;
  }
  if (::strlen(waveFn) == 0)
  {
    usage();
    ret = -1;
    goto zurueck;
  }
  //
  while (1)
  {
    startPos = getNextFn(waveFn, actFn, startPos);
    if (::logging)
      ::printf("next fn=%s\n", actFn);
```

```
    if (startPos < 0)
      break;
    pwf = new WaveFile(actFn);
    cRet = pwf->playFile();
    if (cRet < 0)
    {
      ret = -3;
      goto zurueck;
    }
    ret++;
    delete pwf;
    pwf = NULL;
  }   // end while(1)
  //
  // everything OK here
  zurueck:
  if (pwf != NULL)
    delete pwf;
  if (::logging)
    printf("Play over ret=%d\n", ret);
  return ret;
}   // end main()
```

sample 2: playing threads, callback functions and events

```
// module waveClass3
// Copyright Dr.E.Huckert 07-2011
// Plays a liste of wave files
// Only for Windows (MultiMedia Library winmm.lib)
// This version uses the more complex API calls: waveOutOpen(), waveOutWrite() etc.
// Uses a separate thread for each wave file to be played
// A callback function is used to detect the end of the playing process (WOM_DONE)
// Signals (=events) are sent to indicate the termination of the  threads

// Compile with (Digital Mars C++):
//  dmc -mn -WA -Ae waveClass3.cpp winmm.lib
// For Microsoft: thread safe libraries must be used!

#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <mmsystem.h>

#define MAX_NO_FILES 16

class      WaveFile;
static int logging = 0;

// this class is used by the player threads
class ThreadParams
{
  public:
  char   actFn[256];
  HANDLE event;
};

// classes (structures) used for reading the WAVE files
class MAIN_CHUNK
{
  public:
  DWORD main_chunk;  // must be "RIFF"
  DWORD mlength;     // Laenge ueber alles ohne die ersten 8 Bytes
  DWORD chunk_type;  // must be "WAVE"
};        // 12 bytes
```

```
class FORMAT_CHUNK     // start byte #12
{
  public:
  DWORD sub_chunk;   // muss 'fmt ' sein
  DWORD slength;     // #16
  WORD  format;      // #18
  WORD  channels;    // #20  1=mono, 2=stereo
  DWORD sample_fq;   // #24
  DWORD bytes_per_second;
  WORD  bytes_per_sample;
  WORD  bits_per_sample;
};        // 24 bytes

class DATA_CHUNK     // start at pos. (12 + 24)=36
{
  public:
  DWORD data_type;   // kann 'data' sein oder 'LINK' oder ...
  DWORD data_length;
};      // 8 bytes

// ----------------------------------------------------
// A class to evaluate and play WAVE files.
// class written by EH 01-2015
// modified/enhanced  11-2015
class WaveFile
{
  public:
    int           no_channels;
    int           bits_samp;
    int           bytes_samp;
    unsigned long sample_rate;
    unsigned long bytes_sec;
    unsigned long length;
    WAVEFORMATEX  waveform;
    PWAVEHDR      pWaveHdr;
    char          fileName[256];
    DWORD         callBckParam;
    HWAVEOUT      hWaveOut;
    unsigned char *pData;
    //
  public:
    WaveFile(const char *file_name);  // constructor
    ~WaveFile();
    int readMainChunk(int ifd);
    int readFormatChunk(int ifd);
    long readDataChunks(int ifd);
    int readAttributes(int dataFlag);
    int play(HANDLE event);           // use WaveOut() to play
    int close();
    int open();
};

// ------------------------------------------------------------------
// This function is called by the Windows waveOutXXX() functions when
// certain states are reached (like MM_WOM_DONE when the play is over)
// The message type is the returned in the pInstance parameter. This parameter
// corresponds to the callBckParam variable in class WaveFile (which is thread specific)
void CALLBACK CallBckProcedure(HWAVEOUT  hwo,
                               UINT      message,
                               DWORD     *pInstance, // return data
                               DWORD     *pParam1,
                               DWORD     *param2)
{
  int ret = 0;
  //
  switch (message)
  {
    case MM_WOM_OPEN:
      if (::logging)
        ::printf("CallBckProcedure WOM_OPEN\n");
      ret = 1;
```

```
      break;

    case MM_WOM_DONE:
      if (::logging)
        ::printf("CallBckProcedure WOM_DONE\n");
      ret = 2;
      break;

    case MM_WOM_CLOSE:
      if (::logging)
        ::printf("CallBckProcedure WOM_CLOSE\n");
      ret = 3;
      break;

    default:
      ret = 4;
      break; // do nothing for all other messages
  }   // end switch (message)
  if (pInstance != NULL)
    *pInstance = ret;
  if (::logging)
    ::printf("CallBckProcedure ret=%d\n", ret);
}   // end CallBckProcedure()

// -----------------------------------------
// constructor with a given file name(WAV file)
WaveFile::WaveFile(const char *file_name)
{
  if (file_name == NULL)
    ::strcpy(this->fileName, "");
  else
    ::strcpy(this->fileName, file_name);
  this->no_channels  = 1;
  this->bytes_samp   = 0;
  this->bits_samp    = 0;
  this->length       = 0L;
  this->bytes_sec    = 0L;
  this->sample_rate = 0L;
  this->pWaveHdr     = (PWAVEHDR)new char[sizeof(WAVEHDR)];
  this->callBckParam = 0;
  this->hWaveOut     = 0;
  this->pData        = NULL;
}

// ------------------------------------------------
// Destructor
WaveFile::~WaveFile()
{
  if (this->hWaveOut != 0)
    this->close();
  if (this->pWaveHdr != NULL)
    delete this->pWaveHdr;
  if (this->pData != NULL)
    delete this->pData;
}

// ------------------------------------------------
// closes the sound device
int WaveFile::close()
{
  int ret = 0;
  MMRESULT mmRet;
  //
  if (this->hWaveOut != 0)
  {
    mmRet = ::waveOutClose(this->hWaveOut);
    if (mmRet != MMSYSERR_NOERROR)
    {
      char msg[256];
      ::waveOutGetErrorText(mmRet, msg, sizeof(msg) - 1);
      if (::logging)
```

```
      printf("WaveFile::waveOutClose() reports ERROR: %s\n", msg);
      ret = -1;
    }
    this->hWaveOut = 0;
  }
  if (::logging)
      printf("WaveFile::waveOutClose() ret=%d\n", ret);
  return ret;
}   // end WaveFile::close()

// ------------------------------------------------
// Opens the sound output device
// For multiple simultaneous sounds this must be called for each
// WAVE file - don't close the device too early!
// Returns < 0 upon error
int WaveFile::open()
{
  int ret = 0;
  MMRESULT mmRet;
  //
  // Note: the WAVE device is opened once.
  //            This means that all files to be played must have the same essential
attributes!!!
  //             The code could also be placed  in a static method
  this->waveform.wFormatTag      = WAVE_FORMAT_PCM;
  this->waveform.nChannels       = this->no_channels;
  this->waveform.nSamplesPerSec  = this->sample_rate;
  this->waveform.nAvgBytesPerSec = this->bytes_sec;
  this->waveform.wBitsPerSample  = this->bits_samp;
  this->waveform.nBlockAlign     = (this->bits_samp >> 3) *
                                   this->no_channels;
  this->waveform.cbSize          = 0;  // extra bytes not needed
  if (this->hWaveOut == 0)
  {
    mmRet = ::waveOutOpen(&(this->hWaveOut),
                          WAVE_MAPPER,
                          (WAVEFORMATEX *)(&(this->waveform)),
                          (unsigned long)CallBckProcedure,
                           // important: the API will return in callBckParam the last
message type
                          (unsigned long)(&(this->callBckParam)),
                          CALLBACK_FUNCTION);
    if (mmRet != MMSYSERR_NOERROR)
    {
      char msg[256];
      ::waveOutGetErrorText(mmRet, msg, sizeof(msg) - 1);
      printf("WaveFile::play() waveOutOpen() reports ERROR: %s\n", msg);
      ret = -1;
      goto zurueck;
    }
    if (::logging)
      printf("WaveFile::open() waveOutOpen() finished\n");
    // not sure whether this is needed
    mmRet = ::waveOutReset(this->hWaveOut);
    if (mmRet != MMSYSERR_NOERROR)
    {
      char msg[128];
      ::waveOutGetErrorText(mmRet, msg, sizeof(msg) - 1);
      printf("WaveFile::open() waveOutReset() reports ERROR: %s\n", msg);
      ret = -2;
      goto zurueck;
    }
    if (::logging)
      printf("WaveFile::open() waveOutReset() finished\n");
    ret = 1;
  }  // if (this->hWaveOut == 0)
  //
  zurueck:
  return ret;
}   // end WaveFile::open()
```

```
// ------------------------------------------------------------
// read 12 bytes  main chunk header (WAVE file)
// Returns: the number of bytes read
//          < 0 if error
int WaveFile::readMainChunk(int ifd)
{
  int nb;
  int ret = 0;
  MAIN_CHUNK header;
  //
  nb = ::read(ifd, &header, sizeof(MAIN_CHUNK));
  if (nb != sizeof(MAIN_CHUNK))
  {
    ret = -1;
    goto zurueck;
  }
  if (::strncmp((char *)&(header.main_chunk), "RIFF", 4) != 0)
  {
    ret = -2;
  }
  if (::strncmp((char *)&(header.chunk_type), "WAVE", 4) != 0)
  {
    ret = -3;
  }
  // everything OK here
  ret = nb;
  //
  zurueck:
  if (::logging)
      printf("WaveFile::readMainChunk() ret=%d\n", ret);
  return ret;
}   // end WaveFile::readMainChunk()

// ------------------------------------------------------------
// read the format chunk  (WAVE file)
// Returns: the number of bytes read
//          < 0 if error
int WaveFile::readFormatChunk(int ifd)
{
  int nb;
  int ret = 0;
  FORMAT_CHUNK format;
  //
  nb = ::read(ifd, &format, sizeof(FORMAT_CHUNK));
  if (nb != sizeof(FORMAT_CHUNK))
  {
    ret = -1;
    goto zurueck;
  }
  if (::strncmp((char *)&(format.sub_chunk), "fmt ", 4) != 0)
  {
    ret = -2;
    goto zurueck;
  }
  // This is a "fmt " sub chunk.   transfer the attributes
  this->no_channels = format.channels;
  this->sample_rate = format.sample_fq;
  this->bytes_samp  = format.bytes_per_sample;
  this->bytes_sec   = format.bytes_per_second;
  this->bits_samp   = format.bits_per_sample;
  // everything OK here
  ret = nb;
  //
  zurueck:
  if (::logging)
      printf("WaveFile::readFormatChunk() ret=%d\n", ret);
  return ret;
}   // end WaveFile::readFormatChunk()

// ------------------------------------------------------------
// Read the data chunks, allocate memory for the data  (WAVE file)
```

```
// We evaluate only chunks of type "data" - other chunks are read but
// not evaluated.
// Returns: the number of bytes read
//          < 0 if error
long WaveFile::readDataChunks(int ifd)
{
  long nb;
  long nbTotal = 0L;
  long stored  = 0L;
  long ret = 0;
  DATA_CHUNK data;
  unsigned char *pDest = NULL;
  char buf[1024];
  //
  while (1)
  {
    nb = ::read(ifd, &data, sizeof(DATA_CHUNK));
    if (nb != sizeof(DATA_CHUNK))
    {
      if (nbTotal > 0L)
        break;
      ret = -1;
      goto zurueck;
    }
    nbTotal += nb;
    if (::logging)
    {
      ::memcpy(buf, &(data.data_type), 4);
      buf[4] = 0;
      printf("WaveFile::readDataChunks() type=%s\n", buf);
      printf("WaveFile::readDataChunks() length=%ld\n", data.data_length);
    }
    if (::strncmp((char *)(&(data.data_type)), "data", 4) == 0)
    {
      // note: we reserve space only for the first DATA chunk !!!  - must be changed
      this->length += data.data_length;
      if (this->pData == NULL)
      {
        this->pData = new unsigned char[this->length];
        pDest = this->pData;
      }
    }
    while (data.data_length > 0L)
    {
      nb = sizeof(buf);
      if (nb > data.data_length)
        nb = data.data_length;
      nb = ::read(ifd, buf, nb);
      if (nb <= 0)
      {
        ret = -2;
        goto zurueck;
      }
      nbTotal += nb;
      data.data_length -= nb;
      if (::strncmp((char *)(&(data.data_type)), "data", 4) == 0)
      {
        if ((pDest != NULL) && ((stored + nb) <= this->length))
        {
           // note: store only the first DATA chunk !!!  - must be changed
          ::memcpy(pDest, buf, nb);
          pDest += nb;
          stored += nb;
        }
      }
    }
  }   // end while (1)
  //
  // everything OK here
  ret = nbTotal;
  //
```

```
  zurueck:
  if (::logging)
      printf("WaveFile::readDataChunks() ret=%ld\n", ret);
  return ret;
}   // end WaveFile::readDataChunks()

// ------------------------------------------------
// Open the file, evaluate the attributes.
// Read and keep the data of flag dataFlag is set
int WaveFile::readAttributes(int dataFlag)
{
  int   ret  = 0;
  int   cRet = 0;
  long   lRet = 0L;
  int   ifd  = -1;
  FILE  *stream = NULL;
  char  buff[16];
  int   state = 0;
  long  offset;
  //
  ifd = ::open(this->fileName, O_RDONLY + O_BINARY);
  if (ifd < 0)
  {
    if (::logging)
      printf("WaveFile::readAttributes() ERROR cannot open file=%s\n", this->fileName);
    ret = -1;
    goto zurueck;
  }
  // the  file is here open - read the file header
  cRet = this->readMainChunk(ifd);
  if (cRet < 0)
  {
    // probably wrong file type
    if (::logging)
      printf("WaveFile::readAttributes() ERROR wrong type? file=%s\n", this->fileName);
    ret = -2;
    goto zurueck;
  }
  state = 1;
  //
  // get and keep the essential sound attributes
  cRet = this->readFormatChunk(ifd);
  if (cRet < 0)
  {
    if (::logging)
      printf("WaveFile::readAttributes() ERROR wrong type? file=%s\n", this->fileName);
    ret = -3;
    goto zurueck;
  }
  state = 2;
  if (dataFlag)
  {
    // read and keep the wave data
    state = 3;
    lRet = this->readDataChunks(ifd);
    if (lRet < 0L)
    {
      ret = -4;
      goto zurueck;
    }
  }   // end if (dataFlag)
  //
  // everything OK here
  state = 4;
  if (::logging)
  {
    ::printf("WaveFile::readAttributes() no_channels=%d\n", this->no_channels);
    ::printf("WaveFile::readAttributes() length=%ld\n",     this->length);
    ::printf("WaveFile::readAttributes() sample_rate=%ld\n", this->sample_rate);
    ::printf("WaveFile::readAttributes() bytes_samp=%d\n",  this->bytes_samp);
    ::printf("WaveFile::readAttributes() bytes_sec=%ld\n",  this->bytes_sec);
```

```
    ::printf("WaveFile::readAttributes() bits_samp=%d\n",   this->bits_samp);
  }
  //
  zurueck:
  if (ifd >= 0)
    ::close(ifd);
  if (::logging)
    ::printf("WaveFile::readAttributes() ret=%d state=%d\n",
              ret, state);
  return ret;
}   // end WaveFile::readAttributes()

// ------------------------------------------------
// Thread worker routine
DWORD WINAPI workerRoutine(LPVOID pParams)
{
  int startPos = 0;
  int ret      = 0;
  int cRet;
  WaveFile *pwf = NULL;
  int noFiles  = 0;
  ThreadParams *ptp = NULL;
  //
  if (::logging)
  {
    if (ptp != NULL)
      printf("Thread worker routine started file name=%s\n",
            ptp->actFn);
  }
  ptp = (ThreadParams *)pParams;
  if (ptp == NULL)
  {
    ret = -1;
    goto zurueck;
  }
  try
  {
    pwf  = new WaveFile((const char *)ptp->actFn);
    // start the player
    cRet = pwf->play(ptp->event);
    if (cRet < 0)
    {
      ret = -2;
      goto zurueck;
    }
  }   // end try
  catch (...)
  {
    printf("EXCEPTION in Thread routine workerRoutine()");
    ret = -3;
    goto zurueck;
  }
  //
  zurueck:
  if (::logging)
    printf("thread finished ret=%d\n", ret);
  return ret;
}   // end workerRoutine()

// ------------------------------------------------------------
// Play the file by using API call  ::WaveOut()
int WaveFile::play(HANDLE event)
{
  int      ret = 0;
  int      cRet;
  MMRESULT mmRet;
  DWORD    dword;
  DWORD    dwRepetitions = 1;
  double   noMs = 0.0;
  //
  if (::logging)
```

```
  ::printf("WaveFile::play() file=%s\n", this->fileName);
if (::strlen(this->fileName) == 0)
{
  ret = -1;
  goto zurueck;
}
//
// this call must be executed! Evaluates the attributes, reads and keeps the wave data
cRet = this->readAttributes(1);
if (cRet < 0)
{
  ret = -2;
  goto zurueck;
}
//
this->pWaveHdr->lpData          = (char *)(this->pData);
this->pWaveHdr->dwBufferLength  = this->length;
this->pWaveHdr->dwBytesRecorded = 0;
this->pWaveHdr->dwUser          = 0;
// if only one buffer ist used both flags must be set
this->pWaveHdr->dwFlags         = WHDR_BEGINLOOP |
                                  WHDR_ENDLOOP;
this->pWaveHdr->dwLoops         = dwRepetitions;
this->pWaveHdr->lpNext          = NULL;
this->pWaveHdr->reserved        = 0;
//
// Note: the audio device must be opend for each WAVE file (each thread)
if (this->open() < 0)
{
  ret = -3;
  goto zurueck;
}
//
mmRet = ::waveOutPrepareHeader(this->hWaveOut,
                               this->pWaveHdr,
                               sizeof(WAVEHDR));
if (mmRet != MMSYSERR_NOERROR)
{
  char msg[256];
  ::waveOutGetErrorText(mmRet, msg, sizeof(msg) - 1);
  printf("WaveFile::play() waveOutPrepareHeader() reports ERROR: %s\n", msg);
  ret = -4;
  goto zurueck;
}
//
mmRet = ::waveOutWrite(this->hWaveOut,
                       this->pWaveHdr,
                       sizeof(WAVEHDR));
if (mmRet != MMSYSERR_NOERROR)
{
  char msg[256];
  ::waveOutGetErrorText(mmRet, msg, sizeof(msg) - 1);
  printf("WaveFile::play() waveOutWrite() reports ERROR: %s\n", msg);
  ret = -5;
  goto zurueck;
}
//
while (1)
{
  // stop this loop if message WOM_DONE has been detected
  if (this->callBckParam >= 2)
  {
    ::SetEvent(event);
    break;
  }
  ::Sleep(1);
}   // end while (1)
if (::logging)
  printf("WaveFile::play() callBckParam=%ld\n", this->callBckParam);
//
// Note: we don't close the WAVE device here even if it was opened here
```

```
  //     because it may be reused in other instances of this class
  //
  zurueck:
  if (::logging)
    printf("WaveFile::play() ret=%d\n", ret);
  return ret;
}   // end WaveFile::play()

// -------------------------------------------------------
// This is a tokenizer for the file list in the command line
int getNextFn(char *strFn, char *actFn, int startPos)
{
  int pos  = startPos;
  int i    = 0;
  actFn[0] = 0;
  while (strFn[pos] != 0)
  {
    if (strFn[pos] == ' ')
    {
      pos++;
      continue;
    }
    if (strFn[pos] == '"')
    {
      pos++;
      continue;
    }
    if (strFn[pos] == 0)
      break;
    if (strFn[pos] == ',')
    {
      pos++;
      break;
    }
    actFn[i++] = strFn[pos];
    actFn[i]   = 0;
    pos++;
  }
  if (::strlen(actFn) == 0)
    pos = -1;
  return pos;
}   // end getNextFn()

// -------------------------------------------------------
void usage()
{
  ::printf("usage: waveClass3 -i fileList [-v]\n");
}   // end usage()

// -------------------------------------------------------
int main(int argc, char *argv[])
{
  int      ret = 0;
  int      cRet;
  DWORD    dwRet;
  int      startPos = 0;
  char     fileList[512] = "";
  char     *pActFn;
  ThreadParams *ptp = NULL;
  DWORD    dw_s, wt;
  DWORD    waitTime = 0L;
  HANDLE   s_t;
  HANDLE   evArray[MAX_NO_FILES];
  ThreadParams *threadParamsArray[MAX_NO_FILES];
  int      evArrayCnt = 0;
  //
  for (int n=0; n < MAX_NO_FILES; n++)
  {
    threadParamsArray[n] = NULL;
  }
  // the  filelist is a list of comma separated file names (WAVE files)
```

```
  for (int n=1; n < argc; n++)
  {
    if (::strcmp(argv[n], "-i") == 0)
      ::strcpy(fileList, argv[n + 1]);
    if (::strcmp(argv[n], "-v") == 0)
      ::logging = 1;
  }
  if (::strlen(fileList) == 0)
  {
    usage();
    ret = -1;
    goto zurueck;
  }
  // create an event that should be signalled by the thread to indicate that
  // the play process is finished
  //
  while (1)
  {
    pActFn = new char[256];
    startPos = getNextFn(fileList, pActFn, startPos);
    if (::logging)
      ::printf("main() next fn=%s\n", pActFn);
    if (startPos < 0)
      // no further file name found
      break;
    //
    // Create a separate thread for each file to be played
    // Associate an event with each thread
    // setup the  thread parameters
    ptp = new ThreadParams;
    threadParamsArray[evArrayCnt] = ptp;
    ptp->event = ::CreateEvent(0, false, false, 0);
    evArray[evArrayCnt++] = ptp->event;
    ::strcpy(ptp->actFn, pActFn);
    //
    // start the thread
    s_t = ::CreateThread(NULL,
                         0,
                         workerRoutine,
                         (LPVOID)ptp,
                         0,
                         &dw_s);
    ret++;
  }   // end while(1)
  //
  if (evArrayCnt > 0)
  {
    // wait for all events that are sent when the threads terminate
    dwRet = ::WaitForMultipleObjects((DWORD)evArrayCnt,
                                     evArray,
                                     TRUE,
                                     INFINITE);
    ::Sleep(5);
    if (::logging)
      // dwRet = 0 means: event has been sent, 0x102=Timeout
      printf("main() result waiting on event=%ld\n", (long)dwRet);
  }
  //
  // everything OK here
  zurueck:
  for (int n=0; n < evArrayCnt; n++)
  {
    ptp = threadParamsArray[n];
    if (ptp != NULL)
      delete ptp;
  }
  ::Sleep(5);
  if (::logging)
    printf("main() Play over ret=%d\n", ret);
  return ret;
}   // end main()
```

sample 3: using threads and events only

This is an extract (the *while* loop in member function *play()*) from *waveClass4.cpp* (see the complete code in the ZIP file)*:*

```
// Wait for events until we detect WHDR_DONE in the WAVEHDR structure
// We sent then a termination event to the main process
while (1)
{
  dwRet = ::WaitForSingleObject(this->evHandle, (DWORD)10);
  if (::logging > 1)
    printf("WaveFile::play() event received ret=%ld\n", (long)dwRet);
  if (((this->pWaveHdr)->dwFlags) & WHDR_DONE)
  {
    if (::logging)
      printf("WaveFile::play() WHDR_DONE detected\n");
    // send the termination event to the main process
    ::SetEvent(terminationEvent);
    break;
  }
   // no Sleep() needed!
}   // end while (1)
```

The companion ZIP file

The ZIP file contains:

- *wavedecode.c*: a simple WAVE file decoder, C source
- *wavedecode.exe*: the 32-Bit executable for *wavedecode.c*
- *waveClass3.cpp*: the source code corresponding to sample 2 above
- *waveClass3.exe*: the 32-Bit executable for *waveClass3.cpp*
- *waveClass4.cpp*: the source code that uses only events, no callback functions
- *waveClass4*.exe: the 32-Bit executable for *waveClass4.cpp*
- Four files of 1 sec duration, sinus form. They correspond to the tones c1, g1, e1, a1

Bibliography

Petzold, Charles: Programming Windows 5th ed., Redmond: Microsoft Press, 1999
Kientzle, Tim: A programmer's guide to sound, Reading(Mass.): Addison Wesley, 1997
Scherfgen, David: 3-D Spiele – Programmierung, München: Hanser, 2006