MeMP

„**M**ein **e**infacher **M**p3-**P**layer“

Ein Tutorial zur Programmierung eines Mp3-Players

(Überarbeitet für Version 2.4 der bass.dll)

Inhaltsverzeichnis

---

www.gausi.de
mail@gausi.de

Kapitel 1. Überblick und Vorbereitung

In diesem Tutorial möchte ich einen Einstieg in die Programmierung eines Mp3-Players bieten. Es richtet sich an Delphi-Programmierer, die schon etwas Erfahrung mit Delphi gesammelt haben und sich nun an ein neues, etwas größeres Projekt wagen wollen. Wenn man von einem eigenen Editor oder Taschenrechner die Nase voll hat, dann wird nicht selten ein eigener Player programmiert. Und dann kommen immer wieder dieselben Fragen.

„Wie spiele ich eine mp3 ab?", „Wie kann man automatisch das nächste Lied starten?", „Wie kann ich machen, dass da der Titel steht, und nicht der Dateiname?" oder auch „Wieso bekomme ich ein list index out of bounds (-1)?".

Screenshot unseres fertigen Players. In der Playlist das Album „A tribute to Automatic for the People*"*

Diese Fragen (und ein paar mehr) werden hier erläutert werden. Wir werden von Anfang an darauf achten, dass „da nicht der Dateiname steht". Das mit dem nächsten Titel wird auch klappen. List index out of bounds gibt es nicht, und wir spielen mühelos mp3, ogg, wav, wma, ape, flac und einiges mehr ab. Wir werden Playlisten laden und speichern. Und diese Balken, die bei fast allen Playern rumhüpfen, haben wir dann auch.

Wir werden hier nicht alles selbst entwickeln, sondern „uns auf die Schultern von Giganten stellen". Wir werden z.B. die bass.dll zum Abspielen benutzen. Auch das Auslesen der Informationen („ID3-Tags") aus den verschiedenen Dateien werden wir anderen Klassen überlassen und bereiten diese Informationen nur für uns passend auf.

Delphi startklar machen

Um das Beispiel-Programm zu diesem Tutorial kompilieren zu können, benötigen wir zunächst einmal Delphi. Getestet wurde es mit Delphi 7 Personal und Turbo Delphi Explorer.

Zusätzlich benötigen wir folgendes:

- Die bass.dll. Sie wird das Abspielen der Musik für uns übernehmen. Sie

kann auf http://www.un4seen.com heruntergeladen werden. Aus dem Downloadarchiv benötigen wir am Ende auch einige Teile der Demoprojekte, nämlich die beiden Units `spectrum_vis` und `CommonTypes` aus dem Projekt `SampleVis`.

- Optional Addons zur bass.dll zum Abspielen weiterer Formate wie BassWma, BassFlac, BassApe etc.
- Mp3FileUtils (http://www.gausi.de/delphi/mp3fileutils.php). Diese Unit nimmt uns den Großteil der Arbeit mit mp3-Dateien ab.
- ATL_WmaFile (http://www.gausi.de/delphi/wmatags.php). Eine Unit zum Auslesen der Informationen aus wma-Dateien. Die Version, die bei der Audio-Tools-Library dabei ist, enthält einen kleinen Fehler, kann aber auch benutzt werden. Wer die TNT-Unicode Komponenten nicht hat, muss die TntClasses aus den Uses herausnehmen und den Typ `TTntFileStream = TFileStream;` deklarieren.

Hinweis: Natürlich können anstelle meiner Mp3FileUtils auch andere Units zum Auslesen der ID3-Tags benutzt werden. Dann müssen ggf. einige Stellen im Quellcode angepasst werden. Als mir bekannte Alternativen seien hier die Audio Tools Library (ATL), die Jedis oder die ID3Lib von Muetze1 aus der Delphipraxis genannt. Wie die einzelnen Aufrufe der ID3-Tag-Leseroutinen im Einzelnen genau angepasst werden müssen, sollte man der entsprechenden Dokumentation entnehmen. Ein besonders großer Aufwand dürfte das nicht sein.

In diesem Dokument sind nicht unbedingt alle Methoden der einzelnen Klassen bis ins Detail ausgearbeitet. Das würde das Dokument nur unnötig in die Länge ziehen und dient auch nicht der Übersichtlichkeit. Für eine vollständige Implementierung verweise ich auf das beiliegende Zip-Archiv.

Hinweis zu Delphi 2009

Der Text in diesem Tutorial ist für Delphi 2007 oder früher ausgelegt. Für Delphi 2009 sind auf Grund der Umstellung auf `UnicodeString` einige Anpassungen nötig. Im beiliegenden Quellcode sind an den entsprechenden Stellen Compilerschalter eingefügt, so dass sich der Code mit allen Delphi-Versionen seit Delphi 7 korrekt compilieren lassen sollte. Im Wesentlichen beschränken sich die Änderungen auf das zusätzlich zu verwendende Flag `BASS_UNIOCDE`.

Hinweis zu den Lizenzen

Die bass.dll kann in nicht-kommerziellen Produkten frei verwendet werden, genauere Angaben werden da meines Wissens nach nicht gemacht. Mp3FileUtils steht unter der LGPL, alternativ unter einer BSD-ähnlichen Lizenz. ATL_WmaFile steht wie die gesamte ATL unter der LGPL.

Die in diesem Tutorial entwickelten Klassen stehen unter keiner besonderen Lizenz. Sie können „einfach so" benutzt werden. Ich überlasse es dir, das auf faire Art und Weise zu tun. Bei einem 10.000 Zeilen Programm ist ein Hinweis auf das Tutorial sicherlich kaum noch nötig, bei 1500 Zeilen und damit nur

rund 30% Eigenanteil schon eher.

Hinweise zur bass.dll

Das ursprüngliche Tutorial war für die Version 2.3 der bass.dll ausgelegt. Mit der aktuellen Version 2.4 wurden einige Änderungen eingeführt. Genaueres ist im Changelog der bass.dll nachzulesen – betroffen sind hier im Wesentlichen die Funktionsaufrufe von BASS_ChannelGetAttributes, -SetAttributes, -Getlength, -GetPosition und -SetPosition. Und, ganz wichtig: **Volume ist jetzt ein Float-Typ mit 0=stumm und 1=laut.**

Kapitel 2. Informationen in Audiodateien

Bevor wir mit dem eigentlichen Player beginnen, erstmal etwas Grundlegendes. Wenn ich eine mp3-Datei (oder etwas ähnliches) in einem Player öffne, dann erwarte ich, dass mir der Player anzeigt, wer da gerade singt, und wie das Lied heißt. Auch die Länge des Liedes möchte ich angezeigt bekommen.

Mal ehrlich: Würdet ihr einen Player benutzen, der nichts als den Dateinamen anzeigt? Eben.

Mp3-Dateien

Eine Mp3-Datei besteht aus vielen einzelnen MPEG-Frames – in etwa so, wie ein Film aus vielen Einzelbildern besteht. Jeder dieser Frames besitzt einen 4 Byte großen Header, und in diesen 4 Bytes stecken unter anderem Informationen über Bitrate (z.B. 192 kbit/s), Channelmode (z.B. Stereo) und Samplerate (z.B. 44.1 kHz). Über diese Daten kann man dann die Dauer eines Stückes berechnen.

Informationen wie Titel und Interpret sind im Mp3-Standard eigentlich gar nicht vorgesehen. Hier haben sich aber die so genannten ID3-Tags etabliert und gelten als informeller Standard. In einer ersten Version wurde der ID3v1-Tag ans Ende der Datei geschrieben. Der ID3v1-Tag hat eine feste Größe, kann sehr leicht gefunden und ausgewertet werden, ist aber stark eingeschränkt.

Später wurde der ID3v2-Tag entwickelt, der am Anfang der Datei zu finden ist, eine variable Größe besitzt und nahezu beliebige Informationen aufnehmen kann.

Da Version 1 am Ende und Version 2 am Anfang einer Datei zu finden ist, können natürlich auch beide Versionen in einer Datei auftauchen. Und da die beiden unabhängig voneinander sind, müssen die enthaltenen Informationen auch nicht unbedingt zueinander passen.

ID3-Tags in anderen Formaten

ID3-Tags in anderen Formaten gibt es eigentlich nicht. Bei Ogg Vorbis findet man Vorbis Kommentare, in Wma-Dateien Wma-Tags und bei den Formaten von Apple findet man wieder was anderes. Das schöne an Standards ist ja, dass es so viele davon gibt. Das macht leider ein unkompliziertes Auslesen dieser Daten für alle Formate unmöglich.

Den Anwender interessiert es eigentlich nicht, ob er nun eine mp3-Datei oder eine wma-Datei abspielt. Er möchte einfach wissen, wie das Lied heißt, und wir sollten als Programmierer diesen Wunsch beachten. Und wir werden das so erledigen, dass wir uns einmal darüber Gedanken machen, und in der weiteren Entwicklung nicht mehr. Zu diesem Zweck entwerfen wir unsere erste Klasse, die ein wichtiges Basiselement unseres Players bilden wird.

Die Klasse `TAudioFile`

Unsere Audiodatei-Klasse enthält Properties für Interpret, Titel, Dauer und einige andere Daten, die wir hier der Übersichtlichkeit wegen weglassen. Die Bitrate, Samplerate oder auch das Album wäre sicherlich noch interessant, hat aber zur Klärung des Prinzips keinerlei Bedeutung.

```
TAudioFile = class
  private
    fInterpret: String;
    fTitel    : String;
    fPfad     : String;
    fDauer    : Integer;
    // ...
    procedure GetMp3Info;
    procedure GetWmaInfo;
    procedure SetUnknown;
    function GetPlaylistTitel: String;
  public
    property Interpret: String  read fInterpret  write fInterpret;
    property Titel    : String  read fTitel      write fTitel    ;
    property Pfad     : String  read fPfad       write fPfad     ;
    property Dauer    : Integer read fDauer      write fDauer    ;
    property PlaylistTitel: String read GetPlaylistTitel         ;

    procedure GetAudioInfo(filename: String);
end;
```

Die vorerst einzige öffentliche Methode `GetAudioInfo` ermittelt aus einer Datei diese Informationen – oder versucht es zumindest. Sie ruft dabei je nach Dateityp die passende private Methode auf:

```
procedure TAudioFile.GetAudioInfo(filename: String);
begin
  fPfad := filename;
  if (AnsiLowerCase(ExtractFileExt(filename)) = '.mp3') then
    GetMp3Info
  else
    if AnsiLowerCase(ExtractFileExt(filename)) = '.wma' then
      GetWMAInfo
    else
      SetUnknown;
end;
```

Die einzelnen Prozeduren für das Auslesen der Daten bei einem bestimmten Dateityp greifen schließlich auf die benutzten Units zurück und übertragen anschließend die enthaltenen Informationen auf unser Gerüst. Bei mp3-Dateien sieht das dann zum Beispiel so aus:

```
procedure TAudioFile.GetMp3Info;
var mpegInfo: TMpegInfo;
    ID3v2Tag: TID3V2Tag;
    ID3v1tag: TID3v1Tag;
    Stream: TFileStream;
begin
  // Daten mit MP3FileUtils auslesen
  mpeginfo:=TMpegInfo.Create;
  ID3v2Tag:=TID3V2Tag.Create;
  ID3v1tag:=TID3v1Tag.Create;
  stream := TFileStream.Create(fPfad, fmOpenRead or fmShareDenyWrite);
  id3v1tag.ReadFromStream(stream);
```

```
  stream.Seek(0, sobeginning);
  id3v2tag.ReadFromStream(stream);
  if Not id3v2Tag.exists then
    stream.Seek(0, sobeginning)
  else
    stream.Seek(id3v2tag.size, soFromBeginning);
  mpeginfo.LoadFromStream(Stream);
  stream.free;

  // Daten auf unser Gerüst übertragen
  if mpeginfo.FirstHeaderPosition >- 1 then
  begin
    if id3v2tag.artist <> '' then
      fInterpret := id3v2tag.artist
    else
      fInterpret := id3v1tag.artist;
    if id3v2tag.title <> '' then
      fTitel := id3v2tag.title
    else
      if id3v1tag.title <> '' then
        fTitel := id3v1tag.title
      else
        fTitel := ExtractFileName(fPfad);
    fDauer   := mpeginfo.dauer;
  end else
    SetUnknown;
  MpegInfo.Free;
  Id3v2Tag.Free;
  Id3v1Tag.Free;
end;
```

Der Vorteil an diesem Konstrukt ist recht einfach. Wir können die Ermittlung der Audio-Informationen zu einem späteren Zeitpunkt sehr einfach erweitern, indem wir z.B. eine weitere private Methode `GetOggInfo` schreiben, und die Methode `GetAudioInfo` um einen `else`-Zweig erweitern. Das Prinzip bei diesen Methoden ist immer dasselbe: Suche die Format-spezifischen Informationen aus der Datei zusammen und fülle damit sinnvoll die Felder, die den Anwender am Ende interessieren. Wenn eine Information nicht gefunden werden kann, wird ein Standardwert dafür genommen – z.B. der Dateiname als Titel.

Wir können die Klasse auch durch weitere Eigenschaften erweitern. Zum Beispiel einen String der Form „Interpret – Titel", wie man ihn häufig in Playlisten vorfindet. Anstatt dies immer an Ort und Stelle zu erledigen, schreiben wir einmal einen entsprechenden Getter für diese Property, der auch noch ein paar Fehler ausbügeln kann.

```
function TAudioFile.GetPlaylistTitel: String;
begin
  if Trim(fInterpret) = '' then
    result := fTitel
  else
    result := fInterpret + ' - ' + fTitel;
end;
```

Kapitel 3. MeMP Testprojekt, Version 1.0

Um ein kleines Gefühl für die Anwendung dieser Klasse zu bekommen, bauen wir uns jetzt ein kleines Programm. Wir deklarieren eine globale Variable `GlobalAudioFile` vom Typ `TAudioFile`. Im FormCreate wird sie erzeugt, im OnDestroy wieder freigegeben. Dann packen wir einen Button, einen OpenDialog und ein Memo auf die Form. Im OnClick des Buttons passiert folgendes:

```
procedure TForm1.BtnAuswahlClick(Sender: TObject);
begin
  if AuswahlOpenDialog.Execute then
  begin
    GlobalAudioFile.GetAudioInfo(AuswahlOpenDialog.FileName);
    MemoDateiInfo.Clear;
    MemoDateiInfo.Lines.add('Interpret: ' + GlobalAudioFile.Interpret      );
    MemoDateiInfo.Lines.add('Titel: '     + GlobalAudioFile.Titel          );
    MemoDateiInfo.Lines.add('Pfad: '      + GlobalAudioFile.Pfad           );
    MemoDateiInfo.Lines.add('Dauer: '     + IntToStr(GlobalAudioFile.Dauer));
  end;
end;
```

Wir lassen den Anwender also eine Datei auswählen, lesen die Audio-Informationen aus und listen sie in der Memo auf. Den ganzen komplizierten Kram mit irgendwelchen Tags und Frames haben wir ausgelagert und brauchen uns nicht mehr darum zu kümmern.

Zusammenfassung bis hierhin

Wir haben jetzt eine Grundlage geschaffen, mit der wir prinzipiell beliebige Audiodateien auf die gleiche Art behandeln können. Das Prinzip und der Sinn hinter der Klasse TAudioFile sollte klar sein: Wir brauchen im weiteren Verlauf unseres Projektes nicht mehr ständig darauf zu achten, ob wir eine mp3, wma, ogg, sonstwas Datei haben – wir können immer gleich verfahren, um die Information aus der Datei zu ermitteln

Die einzelnen Methoden und darunter liegenden Dateizugriffsroutinen sind etwas komplizierter und müssen nicht im Detail verstanden worden sein. Wer weiter in die Materie der ID3-Tags einsteigen möchte, der sei auf www.id3.org verwiesen.

Kapitel 4. Abspielen mit der bass.dll

Grundlage für unseren Player bildet die bass.dll. Wer diese noch nicht runtergeladen hat, sollte das jetzt nachholen. Auch wenn diese Dll recht einfach zu benutzen ist, wird das bei komplexeren Playern recht schnell etwas unübersichtlich. Wir packen daher den ganzen Kram rund um die bass in eine eigene Klasse: `TMeMPPlayer`. Diese Klasse wird uns auf bequeme Art und Weise Methoden und Eigenschaften bereitstellen, um unseren Player steuern zu können.

> **Hinweis:** Ich weiß, dass es dafür schon fertige Komponenten gibt. Mit denen habe ich mich noch nicht wirklich beschäftigt, und wir werden sehen, dass das auch nicht unbedingt nötig ist. So furchtbar kompliziert ist die bass.dll dann auch nicht.
>
> Außerdem: Wir können so unseren Player ganz unseren Bedürfnissen anpassen, und sind nicht auf die Fähigkeiten einer externen Komponente angewiesen.

Die Player-Klasse

Das wird jetzt etwas umfangreicher. Daher springen wir direkt ins kalte Wasser und schauen uns mal die Deklaration dieser Klasse im Ganzen an.

```
TMeMPPlayer = class
  private
    fMainStream: DWord;
    fMainVolume: single;
    fMainAudioFile: TAudioFile;

    function MeMP_CreateStream(aFilename: String): DWord;
    procedure SetVolume(Value: single);
    function GetTime: Double;
    function GetProgress: Double;
    procedure SetProgress(Value: Double);
    function GetBassStatus: DWord;
  public
    property Volume:     single read fMainVolume write SetVolume;
    property Time:       Double read GetTime;
    property Progress:   Double read GetProgress write SetProgress;
    property BassStatus: DWord  read GetBassStatus;

    constructor Create;
    destructor Destroy; override;
    procedure InitBassEngine(HND: HWND);
    procedure Play(aAudioFile: TAudioFile);
    procedure Pause;
    procedure Stop;
    procedure Resume;
    procedure StopAndFree;
end;
```

Die Bedeutung der einzelnen Variablen, Methoden und Properties wird im folgenden nach und nach erläutert. Ein Teil sollte aber selbst erklärend sein.

Initialisierung der bass.dll

Neben dem obligatorischen Konstruktor und Destruktor (in denen z.B. `fMainAudioFile` erzeugt bzw. freigegeben werden sollte) ist zunächst die Methode `InitBassEngine` wichtig. Als Parameter erhält sie ein Fenster-Handle. An dieses Handle werden von Seiten der bass.dll auf Wunsch diverse Messages verschickt, was wir hier aber nicht brauchen. Ohne diese Initialisierung würde unser Player später keinen Mucks von sich geben. Sie schlägt fehl, wenn die bass.dll nicht gefunden werden kann, oder wenn sie in einer anderen Version vorliegt.

```
procedure TMeMPPlayer.InitBassEngine(HND: HWND);
begin
  if (HIWORD(BASS_GetVersion) <> BASSVERSION) then
    MessageDLG('Bass 2.4 nicht gefunden', mtError, [MBOK], 0);
  BASS_Init(-1, 44100, 0, HND, nil);
end;
```

Play

Eine der wichtigsten Methoden ist natürlich `Play`. Diese erhält ein AudioFile als Parameter – eben die Datei, die wir abspielen wollen.

```
procedure TMeMPPlayer.Play(aAudioFile: TAudioFile);
begin
  if aAudioFile <> NIL then
  begin
    fMainAudioFile.Assign(aAudioFile);
    StopAndFree;
    fMainstream := MeMP_CreateStream(fMainAudioFile.Pfad);
    BASS_ChannelSetAttribute(fMainStream, BASS_ATTRIB_VOL, fMainVolume);
    BASS_ChannelPlay(fMainStream , True);
  end;
end;
```

Hier wird zunächst das übergebene AudioFile-Objekt kopiert (diese Methode müssen wir unserer AudioFile-Klasse noch hinzufügen). Falls dieses später gelöscht werden sollte (z.B. weil der Anwender die Playlist verändert), ist es im Player immer noch da und verursacht keine Zugriffsverletzungen.

Nachdem wir uns so eine Audiodatei kopiert haben, brechen wir zunächst die laufende Wiedergabe ab. Wir wollen ja immer nur ein Lied gleichzeitig hören.

```
procedure TMeMPPlayer.StopAndFree;
begin
  BASS_ChannelStop(fMainStream);
  BASS_StreamFree(fMainStream);
  fMainStream := 0;
end;
```

Zuletzt erzeugen wir einen neuen Stream, setzen dessen Lautstärke auf den gewünschten Wert und starten die Wiedergabe.

```
function TMeMPPlayer.MeMP_CreateStream(aFilename: String): DWord;
var flags: DWORD;
begin
  if (AnsiLowerCase(ExtractFileExt(aFilename)) = '.mp3') then
    flags := BASS_STREAM_PRESCAN
  else
    flags := 0;
```

```
  result := BASS_StreamCreateFile(False, PChar(aFilename), 0, 0, flags);
end;
```

Wir dröseln hier die Play-Methode in diverse Untermethoden auf, was zunächst nicht unbedingt nötig erscheint. Für Erweiterungen ist es aber durchaus sinnvoll. Man könnte zum Beispiel Parameter einführen, dass die Wiedergabe nicht abrupt beendet wird, sondern langsam ausgeblendet wird. Ebenso könnte man beim Start sanft einblenden. Auch das Erstellen eines Streams bei Webradio sieht etwas anders aus, so dass die Aufteilung sich später als sehr nützlich erweisen kann.

> **Hinweis zu den Flags:** Das Prescan-Flag ist bei mp3-Dateien mit variabler Bitrate nötig. Andernfalls berechnet die bass.dll die Länge des Stückes fehlerhaft, und unsere Fortschrittsanzeige wird durcheinander kommen.
>
> Bei einigen Systemen kann es hier später auch zu seltsamen Effekten kommen. Das lässt sich durch den zusätzlichen Flag BASS_SAMPLE_SOFTWARE beheben, oder durch aktuelle Soundkartentreiber.

Pause, Resume und Stop

Abspielen ist etwas kompliziert, anhalten und weiter fortfahren ganz einfach. Einfach die passende Methode in der Dokumentation der bass suchen und anwenden.

```
procedure TMeMPPlayer.Pause;
begin
  BASS_ChannelPause(fMainStream);
end;

procedure TMeMPPlayer.Stop;
begin
  BASS_ChannelStop(fMainStream);
end;

procedure TMeMPPlayer.Resume;
begin
  BASS_ChannelPlay(fMainStream, False);
end;
```

Auch hier könnte man das Ganze durch sanftes Ein- oder Ausblenden aufwerten. Als Suchbegriff für die Bass-Dokumentation sei an dieser Stelle nur `BASS_ChannelSlideAttributes` genannt.

Time, Progress und Volume

Auch nichts wildes dabei, der Vollständigkeit halber führen wir die Getter und Setter hier auch an. Nichts weiter als etwas Fehlerkorrektur und Aufrufen der passenden Bass-Funkionen. Ein Getter für die Lautstärke ist dabei nicht notwendig – da lesen wir einfach die private Variable `fMainVolume` aus.

```
procedure TMeMPPlayer.SetVolume(Value: single);
begin
  if Value < 0 then Value := 0;
  if Value > 1 then Value := 1;
  fMainVolume := Value;
  BASS_ChannelSetAttribute(fMainStream, BASS_ATTRIB_VOL, fMainVolume);
end;
```

```
function TMeMPPlayer.GetTime: Double;
begin
  if (fMainStream <> 0) then
    result := BASS_ChannelBytes2Seconds(fMainStream,
                BASS_ChannelGetPosition(fMainStream, BASS_POS_BYTE))
  else
    result := 0;
end;

function TMeMPPlayer.GetProgress: Double;
begin
  if (fMainStream <> 0) then
    result := BASS_ChannelGetPosition(fMainStream, BASS_POS_BYTE) /
                BASS_ChannelGetLength(fMainStream, BASS_POS_BYTE)
  else
    result := 0;
end;

procedure TMeMPPlayer.SetPosition(Value: Longword);
begin
  BASS_ChannelSetPosition(fMainStream, Value, BASS_POS_BYTE);
end;

procedure TMeMPPlayer.SetProgress(Value: Double);
begin
  if Value < 0 then Value := 0;
  if Value > 1 then Value := 1;
  SetPosition(Round(BASS_ChannelGetLength(fMainStream, BASS_POS_BYTE) * Value));
end;
```

Der Unterschied zwischen Time und Progress sollte eigentlich klar sein. In unserem fertigen Player wollen wir später eine Anzeige haben, an welcher Stelle wir uns im Lied befinden. Dies soll zum einen durch ein Label erfolgen, an dem die aktuelle Zeit abzulesen ist. Zum anderen wollen wir auch eine Art Scrollleiste haben, die uns den relativen Fortschritt anzeigt.

Einen Setter für die Zeit brauchen wir vorerst nicht (es sei denn, wir wollen später eine Funktion wie „Springe zum Anfang der dritten Minute" realisieren). Den Fortschritt zu ändern, ist aber eine Standard-Funktion in jedem Player. Man packt das sich bewegende Dingens im Fenster mit der Maus an, und schiebt es ein Stückchen nach vorne oder nach hinten.

Status

Zuletzt eine kleine Mini-Funktion, die uns den Status der Bass-Engine nach außen liefert. Hierüber können wir erfahren, ob der Player gerade abspielt, pausiert oder gestoppt ist.

```
function TMeMPPlayer.GetBassStatus: DWord;
begin
  result := BASS_ChannelIsActive(fMainStream);
end;
```

Kapitel 5. MeMP Testprojekt, Version 2.0

Wir erweitern nun unser erstes kleines Projekt aus Kapitel 3. Wir fügen zwei Buttons für Play und Stop ein, zwei ScrollBars für die Abspielposition und Lautstärke sowie einen Timer und zwei Label. Der Abspielbutton wird auch gleichzeitig als Pausebutton fungieren. Damit das vernünftig funktioniert, fragen wir im OnClick-Event des Playbuttons den Status des Players ab.

```
procedure TForm1.BtnPlayPauseClick(Sender: TObject);
begin
  case MeMPPlayer.BassStatus of
    BASS_ACTIVE_STOPPED:
      begin
        MeMPPlayer.Play(GlobalAudioFile);
        LblTitel.Caption := GlobalAudioFile.PlaylistTitel;
      end;
    BASS_ACTIVE_PLAYING: MeMPPlayer.Pause;
    BASS_ACTIVE_PAUSED : MeMPPlayer.Resume;
  end;
end;
```

Wenn der Player gerade abspielt, stellen wir ihn auf Pause, wenn er auf Pause steht, fahren wir mit der Wiedergabe fort, und wenn er gestoppt ist, dann starten wir die Wiedergabe mit der zuletzt ausgewählten Datei.

Die einzelnen Konstanten sind in der bass.pas definiert. Da wir diese in der Hauptunit nicht brauchen (dafür haben wir ja gerade die Player-Klasse geschrieben) können wir sie aber auch hier nochmal deklarieren:

```
Const
  BASS_ACTIVE_STOPPED = 0;
  BASS_ACTIVE_PLAYING = 1;
  BASS_ACTIVE_PAUSED  = 3;
```

Natürlich gehört ins OnCreate der Form die Erstellung und Initialisierung des Players, beim Beenden sollte man ihn wieder freigeben.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  GlobalAudioFile := TAudioFile.Create;
  MeMPPlayer := TMeMPPlayer.Create;
  MeMPPlayer.InitBassEngine(Handle);
  MeMPPlayer.Volume := 1;
end;
```

Timer

Den Timer brauchen wir zur Fortschrittsanzeige. In relativ kurzen Intervallen (ein Wert zwischen 100 und 500ms sollte reichen) fragen wir beim Player die gegenwärtige Abspielposition und den Fortschritt ab und übertragen dies auf die Form. Da wir von unserer Klasse die Zeit in Sekunden erhalten, rechnen wir ein bisschen, damit wir das in der gewohnten Art und Weise angezeigt bekommen.

```
procedure TForm1.MainTimerTimer(Sender: Tobject);
var t: integer;
begin
```

```
  MainScrollbar.Position := Round(MeMPPlayer.Progress * 100);
  t := Round(MeMPPlayer.Time);
  LblTime.Caption := Format('%.2d:%.2d',[t Div 60, t Mod 60])
end;
```

Die ScrollBars

Wenn der Anwender an den beiden ScrollBars rumspielt, setzen wir die Position oder Lautstärke entsprechend neu. Wir gehen hier davon aus, das beide Leisten einen Maximalwert von 100 und Minimalwert von 0 haben.

```
procedure TForm1.MainScrollbarScroll(Sender: TObject;
  ScrollCode: TScrollCode; var ScrollPos: Integer);
begin
  MeMPPlayer.Progress := ScrollPos / 100;
end;

procedure TForm1.VolumeScrollbarScroll(Sender: TObject;
  ScrollCode: TScrollCode; var ScrollPos: Integer);
begin
  MeMPPlaylist.Volume := Scrollpos / 100;
end;
```

Zusammenfassung bis hierhin

Wir haben jetzt einen kleinen Player, der einzelne Dateien abspielen kann. WMA-Dateien kann er noch nicht, dass kommt später. Aber der Player zeigt die Zeit und den Fortschritt an, wir können letzteren leicht ändern und die Lautstärke anpassen.

Kapitel 6. Events

Wenn wir jetzt auf den Play-Button klicken, dann fängt die Wiedergabe an, falls wir zuvor eine Datei ausgewählt haben. Allerdings steht dann auf dem Button weiterhin „Play", obwohl er dann ja eigentlich die Funktion „Pause" besitzt. Wir könnten nun im OnClick-Event das Ändern der Caption einfügen. Und wenn wir dann auf Stop klicken, muss die Caption des Play-Buttons wieder zurück auf Play gesetzt werden.

Wenn man den Player später ausbauen möchte, verliert man so schnell den Überblick, und wenn man an einer Stelle vergisst, den Code zu ändern, führt das für den User schnell zu Inkonsistenzen und sehr unschönen Effekten. Und für uns als Programmierer zu einem unglaublichen Chaos.

Wir fügen unserer Player-Klasse daher einige Events hinzu, die bei passender Gelegenheit abgefeuert werden, und auf die wir in unserer Anwendung dann entsprechend reagieren können.

```
TMeMPPlayer = class
  private
    fOnPlay    : TNotifyEvent;
    fOnPause   : TNotifyEvent;
    fOnResume  : TNotifyEvent;
    fOnStop    : TNotifyEvent;
    fOnChange  : TNotifyEvent;
    fOnEndFile : TNotifyEvent;
    fFileEndSyncHandle: DWord;
    procedure SetEndSync;
  public
    property OnPlay    : TNotifyEvent read fOnPlay     write fOnPlay    ;
    property OnPause   : TNotifyEvent read fOnPause    write fOnPause   ;
    property OnResume  : TNotifyEvent read fOnResume   write fOnResume  ;
    property OnStop    : TNotifyEvent read fOnStop     write fOnStop    ;
    property OnChange  : TNotifyEvent read fOnChange   write fOnChange  ;
    property OnEndFile : TNotifyEvent read fOnEndFile  write fOnEndFile ;
end;
```

OnPlay, OnPause, ...

Diese Events werden ausgelöst, wenn sich der Status des Players ändert. Unsere Pause-Methode sieht dann z.B. so aus:

```
procedure TMeMPPlayer.Pause;
begin
  BASS_ChannelPause(fMainStream);
  if assigned(fOnPause) then fOnPause(Self);
end;
```

Die Methoden Play, Stop und Resume müssen entsprechend erweitert werden.

OnChange

Dieses Event wird zusätzlich zu OnPlay ausgelöst, um z.B. eine Aktualisierung der Titelanzeige vornehmen zu können. Zu diesem Zweck führen wir auch eine neue Property ein, die einfach den Playlistentitel des zugrunde liegenden

Audiofiles weiterleitet.

```
TMeMPPlayer = class
  private
    function GetPlaylistTitel: String;
  public
    property PlaylistTitel: String read GetPlaylistTitel;
end;

function TMeMPPlayer.GetPlaylistTitel: String;
begin
  if assigned(fMainAudioFile) then
    result := fMainAudioFile.PlaylistTitel
  else
    result := '-';
end;
```

OnEndFile

Wir wollen ja später einen Player haben, der nicht nur ein Lied abspielt, sondern im Anschluss daran direkt das nächste. Die bass.dll bietet einem nun die Möglichkeit, bei Ende eines Liedes benachrichtigt zu werden. Dies geschieht über so genannte Synchronizer, die nach Erzeugen des Streams diesem hinzugefügt werden können.

```
procedure TMeMPPlayer.SetEndSync;
begin
  if Assigned(fOnEndFile) then
    fFileEndSyncHandle := Bass_ChannelSetSync(fMainStream,
                  BASS_SYNC_END, 0, @EndFileProc, Self);
end;
```

Wir fügen hier unserer Wiedergabe einen Synchronizer vom Typ `BASS_SYNC_END` hinzu. Wenn es soweit ist, wird die Prozedur `EndFileProc` ausgeführt, die als Parameter unter anderem das aufrufende Player-Objekt erhält. Diese feuert dann einfach das `OnEndFile` Event ab:

```
procedure EndFileProc(handle: HWND; Channel, Data, User: DWord); stdcall;
begin
  TMeMPPlayer(User).OnEndFile(TMeMPPlayer(User));
end;
```

Die Aufrufkonvention **`stdcall`** ist hierbei wichtig. Auf ähnliche Art könnte man auch einen Synchronizer setzen, der uns benachrichtigt, wenn wir *fast* am Ende des Stückes sind. In so einem Fall könnten wir die laufende Wiedergabe sanft ausblenden, die nächste einblenden und so einen schöneren Übergang zwischen einzelnen Titeln zu erhalten.

Kapitel 7. MeMP Testprojekt, Version 3.0

Wir werden nun die Events benutzen, um den Player etwas schöner zu gestalten. Dafür schreiben wir fünf kurze Prozeduren.

```
procedure TForm1.OnMeMPPlay(Sender: TObject);
begin
  MainTimer.Enabled := True;
  BtnPlayPause.Caption := 'Pause';
end;

procedure TForm1.OnMeMPPause(Sender: TObject);
begin
  MainTimer.Enabled := False;
  BtnPlayPause.Caption := 'Weiter';
end;

procedure TForm1.OnMeMPStop(Sender: TObject);
begin
  MainTimer.Enabled := False;
  BtnPlayPause.Caption := 'Play';
end;

procedure TForm1.OnMeMPChange(Sender: TObject);
begin
  LblTitel.Caption := (Sender as TMeMPPlayer).PlaylistTitel;
end;

procedure TForm1.OnMeMPEndFile(Sender: TObject);
begin
  MeMPPlayer.Play(GlobalAudioFile);
end;
```

Den Abspielbutton beschriften wir je nach Situation um und schalten den Timer an oder aus. Wenn ein Lied zu Ende ist, starten wir die Wiedergabe erneut. Wurde in der Zwischenzeit kein anderes Lied ausgewählt, wiederholt sich das alte, ansonsten startet das neue. Es wird halt das abgespielt, was in der Memo steht.

Diese Methoden weisen wir im FormCreate den entsprechenden Events unserer Player-Instanz zu. Im Falle von Resume machen wir dasselbe wie bei Play – so ein gewaltiger Unterschied ist da ja nun auch nicht.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // ...
  MeMPPlayer.OnEndFile := OnMeMPEndFile;
  MeMPPlayer.OnPlay    := OnMeMPPlay;
  MeMPPlayer.OnResume  := OnMeMPPlay;
  MeMPPlayer.OnPause   := OnMeMPPause;
  MeMPPlayer.OnStop    := OnMeMPStop ;
  MeMPPlayer.OnChange  := OnMeMPChange;
end;
```

Kapitel 8. Das Plugin-System der bass.dll

Die bass.dll unterstützt von sich aus nur recht wenige Formate. Die gängigsten wie mp3, wav und ogg sind dabei, aber schon bei wma gibt es Probleme – das geht einfach nicht. Allerdings gibt es dafür Zusatz-Dlls, die diese Aufgabe übernehmen können. Es gibt eine für wma, eine für ape, flac, aac und einige weitere.

Zunächst einmal müsste jedes Format über die entsprechende `BASS_XXX_StreamCreateFile`-Methode gestartet werden, was zu einer ähnlichen Konstruktion wie beim Auslesen der Audio-Informationen führen würde. Das würde aber bedeuten, dass wir als Programmierer *alle* dlls direkt mitliefern müssen, und auch alle zugehörigen .pas-Dateien in unser Projekt aufnehmen müssen. Und spätestens bei der bass_aac.dll bekommen wir hier Probleme, denn diese steht unter der GPL (und ich gehe jetzt einfach mal davon aus, dass wir unseren Player nicht unbedingt unter dieser Lizenz vertreiben wollen).

An dieser Stelle verweise ich mal auf die Beispiel-Projekte, die bei der bass.dll mitgeliefert werden. Da stecken nämlich eine Menge nützlicher Dinge drin, und man kann sich da einiges abgucken. Jetzt brauchen wir Plugins.

Die bass.dll bietet nämlich ein recht einfaches Plugin-System, um genau diese Probleme zu beheben. Wir können beim Start unseres Players nach gültigen Plugin-Dlls in unserem Programmverzeichnis suchen und diese einbinden. Wir bekommen dabei auch Informationen darüber, was dieses Plugin kann, z.B. welche Dateitypen unterstützt werden. Die bass.dll wiederum sucht sich dann beim Abspielen einer Datei automatisch das passende Plugin heraus, und nimmt uns so eine Menge Arbeit ab.

Der User hat davon den Vorteil, dass er sich bei Bedarf die passende Dll besorgen kann, um mit unserem Player auch das von ihm bevorzugte etwas exotischere Format abspielen zu können.

Wir erweitern die Player-Klasse also noch ein wenig.

```
TMeMPPlayer = class
  private
    fFilter: String;
    fValidExtensions: Tstringlist;
  public
    property Filter: String read fFilter;
    procedure InitPlugins(PathToDlls: String);
    function IsPlayableFile(aFilename: String): Boolean;
end;
```

Die Eigenschaft Filter können wir später bei einem OpenDialog einsetzen, die Funktion `IsPlayableFile` sagt uns, ob die bass.dll mit einer Datei, die so eine Endung besitzt, etwas anfangen kann. Dies können wir später z.B. bei einem OpenDirectoryDialog anwenden, wenn wir den ausgewählten Ordner (rekursiv) nach verwertbaren Musikdateien durchsuchen, oder per Drag&Drop einige Dokumente auf die Form ziehen. Der Filter und die Liste der gültigen Formate

werden in der Prozedur `InitPlugins` ermittelt.

```
procedure TMeMPPlayer.InitPlugins(PathToDlls: String);
var fh: THandle;
    fd: TWin32FindData;
    Plug: DWORD;
    Info: PBass_PluginInfo;
    a,i: integer;
    tmpext: TStringlist;
begin
  PathToDlls := IncludeTrailingPathDelimiter(PathToDlls);
  fFilter := '|Standardformate (*.mp3;*.mp2;*.mp1;*.ogg;*.wav;*.aif)'
             + '|*.mp3;*.mp2;*.mp1;*.ogg;*.wav*;*.aif';
  // Plugins laden. Code aus dem Plugin-Beispiel-Projekt
  fh := FindFirstFile(PChar(PathToDlls + 'bass*.dll'), fd);
  if (fh <> INVALID_HANDLE_VALUE) then
  try
    repeat
      Plug := BASS_PluginLoad(fd.cFileName, 0);
      if Plug <> 0 then
      begin
        // Plugin-Info ermitteln
        Info := BASS_PluginGetInfo(Plug);
        for a := 0 to Info.formatc - 1 do
        begin
          // Filter erweitern
          fFilter := fFilter + '|' + Info.Formats[a].name + ' ('
                     + Info.Formats[a].exts + ')|' + Info.Formats[a].exts;
          //einzelne Erweiterungen ermitteln
          tmpext := Explode(';',Info.Formats[a].exts);
          for i := 0 to tmpext.Count - 1 do
            fValidExtensions.Add(StringReplace(tmpext.Strings[i],'*', '',[]));
          FreeAndNil(tmpext);
        end;
      end;
    until FindNextFile(fh, fd) = false;
  finally
    Windows.FindClose(fh);
  end;
end;
```

Diese Prozedur ist etwas länger, aber prinzipiell einfach. Das übergebene Verzeichnis wird nach Dateien durchsucht, die ein Plugin für die bass.dll sein könnten. Die Funktion `BASS_PluginInfo` liefert diverse Informationen zurück, die wir in den Filter und die Liste der gültigen Datenamenserweiterungen eintragen.

Die zusätzliche Funktion IsPlayableFile überprüft später einfach, ob der übergebene Dateiname eine Erweiterung besitzt, die in dieser Liste vorkommt.

```
function TMeMPPlayer.IsPlayableFile(aFilename: String): Boolean;
begin
  result := fValidExtensions.IndexOf(ExtractFileExt(aFilename)) >= 0;
end;
```

Da die Initialisierungsfunktion nur die Erweiterungen aus den Plugins bestimmt, sollten wir im Konstruktor des Players die Standardformate einfügen.

Kapitel 9. MeMP Testprojekt, Version 4.0

Wir bauen nun noch das Plugin-System in unseren Player ein. Nach der Initialisierung der bass.dll laden wir die Plugins

```
MeMPPlayer.InitPlugins(ExtractFileDir(ParamStr(0)));
```

und das OnClick-Event des Auswahl-Buttons ändern wir so ab:

```
if AuswahlOpenDialog.Execute then
  begin
    if MeMPPlayer.IsPlayableFile(AuswahlOpenDialog.FileName) then
    begin
      GlobalAudioFile.GetAudioInfo(AuswahlOpenDialog.FileName);
      MemoDateiInfo.Clear;
      // (Memo mit neuen Infos füllen...)
    end else
      Showmessage('Die ausgewählte Datei kann nicht abgespielt werden.');
  end;
```

Zusammenfassung bis hierhin

Unser Player fängt nun bereits einige Fehler des Nutzers ab. Wenn die Dateiendung nicht stimmt, wird das Abspielen gar nicht erst versucht. Während die Wiedergabe läuft, kann eine neue Datei ausgewählt werden, die als nächstes abgespielt wird.

Dabei ist das Programm so flexibel, dass es vom Anwender nachträglich erweitert werden kann, indem er die entsprechenden Zusatzdateien für die bass.dll in das Programmverzeichnis kopiert.

Kapitel 10. Eine Playlist

Den wirklich komplizierten Teil haben wir jetzt hinter uns. Wir leiten jetzt von unserer Player-Klasse eine Playlist-Klasse ab, die eine Liste von Dateien verwalten und abspielen kann. Mit der ganzen Vorarbeit, die wir bis jetzt geleistet haben, ist das nicht mehr schwer. Und keine Angst: Das wird die letzte Klasse, die wir erstellen. Eine Instanz dieser Klasse wird dann den Hauptteil unseres Players bilden.

```
TPlaylistAddItemEvent
          = procedure(Sender: TObject; NewAudioFile: TAudioFile) of Object;

TMeMPPlaylist = Class(TMeMPPlayer)
  private
    fPlayingIndex: Integer;
    fPlayingFile: TaudioFile;
    fPlaylist: TObjectlist;
    // einige Events
    fOnClear   : TNotifyEvent;
    fOnAddItem : TPlaylistAddItemEvent;

    function GetNextAudioFileIndex: Integer;
    function GetPrevAudioFileIndex: Integer;
  public
    property PlayingFile: TAudioFile read fPlayingFile;
    property PlayingIndex: Integer read fPlayingIndex;
    property OnClear: TNotifyEvent read fOnClear write fOnClear;
    property OnAddItem: TPlaylistAddItemEvent read fOnAddItem write fOnAddItem;

    constructor Create;
    destructor Destroy; override;
    procedure Add(aAudiofileName: String);
    procedure Play(aIndex: Integer); Overload;
    procedure PlayNext;
    procedure PlayPrevious;
    procedure Clear;
  end;
```

Wir haben hier direkt zwei weitere Events eingefügt, die wir benötigen. Die beiden anderen Properties geben uns Auskunft darüber, an welcher Stelle wir uns in der Playlist befinden, bzw. welche Datei gerade abgespielt wird.

> **Hinweis:** PlayingFile entspricht meist dem fMainAudioFile der zugrunde liegenden Playerklasse. Der Unterschied besteht darin, dass Playingfile nur eine Referenz auf ein Objekt in der Liste ist, um z.B. den Eintrag dazu in einer Listview anders zu zeichnen. Mit der Trennung wird es möglich sein, die Playlist zu leeren, ohne ggf. die laufende Wiedergabe abzubrechen. In dem Fall kennt der Player noch die Datei (der hat sie sich ja kopiert), die Playlist hat sie aber schon „vergessen". Wir müssen bei Erweiterungen der Klasse darauf achten, dass PlayingFile immer auf ein gültiges Objekt verweist oder NIL ist. Es ist sinnvoll, bei der Erweiterung der Klasse darauf zu achten, dass PlayingFile und PlayingIndex immer so weit wie möglich konsistent sind.

Im Konstruktor erzeugen wir die interne ObjectList und initialisieren das PlayingFile mit Nil. Im Destruktor räumen wir wieder auf.

Die Methode `Add` erstellt aus dem übergebenen Dateinamen ein Objekt der

Klasse TAudioFile, ermittelt die Informationen aus dieser Datei, fügt es der Liste hinzu und löst dann das `OnAddItem`-Event aus.

```
procedure TMeMPPlaylist.Add(aAudiofileName: String);
var NewFile: TAudioFile;
begin
  NewFile := TAudioFile.Create;
  NewFile.GetAudioInfo(aAudiofileName);
  fPlaylist.Add(NewFile);
  if assigned(fOnAddItem) then
    fOnAddItem(Self, NewFile);
end;
```

Zum Abspielen übergeben wir einfach nur den Index der Datei, die wir abspielen wollen. Wir erlauben an dieser Stelle auch einen negativen Index. In diesem Fall wird das zuletzt abgespielte Stück erneut abgespielt. Die Bedingungen davor sollten sicherstellen, dass Fehleingaben abgefangen werden.

```
procedure TMeMPPlaylist.Play(aIndex: Integer);
begin
  if aIndex >= 0 then
    fplayingIndex := aIndex;
  if (fplayingIndex > fPlaylist.Count - 1) and (fPlaylist.Count > 0) then
    fplayingIndex := fPlaylist.Count - 1;
  if (fplayingIndex >= 0) and (fPlayingIndex < fPlaylist.Count) then
    fPlayingFile := TAudioFile(fPlaylist[fplayingIndex]);
  play(fPlayingFile);
end;
```

PlayNext und PlayPrevious sind schnell erklärt – es wird einfach das nächste bzw. vorherige Stück in der Liste abgespielt.

```
procedure TMeMPPlaylist.PlayNext;
begin
  Play(GetNextAudioFileIndex);
end;

procedure TMeMPPlaylist.PlayPrevious;
begin
  Play(GetPrevAudioFileIndex);
end;
```

Die hier aufgerufenen privaten Methoden `GetNextAudioFileIndex` und `GetPrevAudioFileIndex` liefern im einfachsten Fall einfach `fPlayingIndex` +/- 1 zurück. Wir könnten hier auch optional eine Zufallswiedergabe einbauen.

Zuletzt die Methode Clear, die den Player anhält und die Playlist in den ursprünglichen Zustand zurückversetzt.

```
procedure TMeMPPlaylist.Clear;
begin
  Stop;
  fPlaylist.Clear;
  if Assigned(fOnClear) then
    fOnClear(Self);
  fPlayingFile := Nil;
  fPlayingIndex := 0;
end;
```

Kapitel 11. MeMP, Version 0.0,5

Bauen wir uns jetzt als unseren ersten halbwegs richtigen Player zusammen. Wir können auch unser letztes Projekt fortführen, was wir aber gründlich entrümpeln. Zunächst schmeißen wir die Memo von der Form runter, ebenso den ganzen Code drumherum. Stattdessen packen wir eine ListView auf die Form, stellen den Style auf vsReport und fügen zwei Spalten hinzu, Titel und Dauer.

Bei unserem OpenDialog stellen wir ofAllowMultiSelect auf True und erlauben so die Auswahl mehrerer Dateien. Caption und Name von Button und Dialog werden der neuen Funktion „Dateien der Playlist hinzufügen" angepasst.

Als nächstes entfernen wir den `MeMPPlayer` und ersetzen ihn durch das erweiterte Konstrukt `MeMPPlaylist`. Die globale Variable `GlobalAudioFile` wird nun auch nicht mehr benötigt.

Hinzu kommen zwei Buttons für „Vor" und „Zurück".

Die FormCreate-Prozedur erweitern wir, um die Playlist komplett zu initialisieren.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MeMPPlaylist := TMeMPPlaylist.Create;
  // ...
  MeMPPlaylist.OnAddItem  := OnPlaylistAddItem;
  MeMPPlaylist.OnClear    := OnPlaylistClear;

  AddFilesDialog.Filter := MeMPPlaylist.Filter;
end;
```

Das MeMPPlayer.Play im OnClick-Event des Play-Buttons ersetzen wir durch

```
MeMPPlaylist.Play(-1);
```

und unsere Reaktion auf das Event `OnEndFile` ändern wir so ab:

```
procedure TForm1.OnMeMPEndFile(Sender: TObject);
begin
  MeMPPlaylist.PlayNext;
end;
```

Der alte Auswahl-Button bekommt nun diese Funktion:

```
procedure TForm1.BtnAddFilesClick(Sender: TObject);
var i: Integer;
begin
  if AddFilesDialog.Execute then
    for i := 0 to AddFilesDialog.Files.Count - 1 do
      MeMPPlaylist.Add(AddFilesDialog.Files[i]);
end;
```

Damit die eingefügten Dateien auch in der Listview angezeigt werden, reagieren wir entsprechend auf das OnAddItem-Event der Playlist.

```
procedure TForm1.OnPlaylistAddItem(Sender: TObject; NewAudioFile: TAudioFile);
var newItem: TListItem;
    t: Integer;
begin
```

```
  newItem := PlaylistView.Items.Add;
  newItem.Caption := NewAudioFile.PlaylistTitel;
  t := NewAudioFile.Dauer;
  NewItem.SubItems.Add(Format('%d:%.2d',[t Div 60, t Mod 60]));
  NewItem.Data := NewAudioFile;
end;
```

Damit wir den Player halbwegs schön bedienen können, kommt ein

```
procedure TForm1.PlaylistViewDblClick(Sender: TObject);
begin
  MeMPPlaylist.Play(PlaylistView.ItemIndex);
end;
```

dazu, und die Anzeige, welches Lied gerade in der Liste aktuell ist, erledigen wir einfach so:

```
procedure TForm1.PlaylistViewCustomDrawItem(Sender: TCustomListView;
  Item: TListItem; State: TCustomDrawState; var DefaultDraw: Boolean);
begin
  If TAudioFile(Item.Data) = MeMPPlaylist.PlayingFile then
    PlaylistView.Canvas.Font.Style := [fsBold];
end;
```

Damit das vernünftig funktioniert, müssen wir im OnPlay-Event ein

```
PlaylistView.Refresh;
```

einfügen, damit bei einem Titelwechsel die Anzeige aktualisiert wird. Möchte man z.B. auch ein Statusicon davor setzen, sollte man das auch im OnPause, OnResume und OnStop hinzufügen.

Bleiben noch die Buttons „Vor" und „Zurück" sowie das OnClear-Event der Playlist. Aber was da zu tun ist, ist ja wohl hoffentlich klar, oder?

Zusammenfassung bis hierhin

Das Konzept für unseren Player ist nun fertig. Wir haben eine Playlist, die neben dem Titel auch die Spieldauer anzeigt, die Playlist läuft von vorne bis hinten durch, wir können stoppen, pausieren und weiterspielen, die Lautstärke ändern und im Titel an eine beliebige Stelle springen. Das ist schon eine ganze Menge. Dabei haben wir darauf geachtet, dass wir eine klare Trennung von Daten und Darstellung haben, und somit unser Programm leicht erweiterbar ist, ohne alles umzuschreiben.

Im Zusammenhang mit Playlisten ist noch eine Frage sehr interessant, nämlich das Speichern und Laden der gängigen Formate, und irgendwie ist ein Player ohne eine Visualisierung auch nicht wirklich fertig. Man muss ja nicht direkt psychedelischen Unsinn darstellen, aber so ein paar hüpfende Balken sind doch ganz nett. Und das tollste daran: Das geht ganz einfach. Sagte ich schon, dass man sich die Beispielprojekte der bass.dll mal genauer ansehen sollte? Dann ist ja gut – also auf geht's zum Endspurt!

Kapitel 12. Playlisten laden und speichern

Playlist-Formate gibt es ungefähr genauso viele wie Player. Ein recht einfaches und noch dazu sehr verbreitetes ist m3u, das u.a. von Winamp gerne verwendet wird. Wir erweitern jetzt unsere Playlistklasse um Lade- und Speicherroutinen für solche Dateien.

```
TMeMPPlaylist = Class(TMeMPPlayer)
  private
    procedure LoadFromM3U(aFilename: String);
    procedure SaveAsM3U(aFilename: String);
  public
    procedure LoadFromFile(aFilename: String);
    procedure SaveToFile(aFilename: String);
end;
```

Das Vorgehen ist hier ähnlich wie ganz am Anfang bei der Audioklasse. Wir haben eine öffentliche Methode, die je nach Dateiendung die richtige Untermethode aufruft. Wer mag, kann später Methoden zum Laden und Speichern anderer Formate hinzufügen. Für Webradio ist das pls-Format noch interessant, was im Wesentlichen eine Ini-Datei ist.

```
procedure TMeMPPlaylist.LoadFromFile(aFilename: String);
begin
  if LowerCase(ExtractFileExt(aFilename)) = '.m3u' then
    LoadFromM3U(aFilename);
end;

procedure TMeMPPlaylist.SaveToFile(aFilename: String);
begin
  if LowerCase(ExtractFileExt(aFilename)) = '.m3u' then
    SaveAsM3U(aFilename);
end;
```

M3U-Listen kann man z.B. wie folgt laden und für unsere Zwecke verarbeiten. Da wir sowieso die Dateien untersuchen, ignorieren wir die oft vorhandenen ExtInf-Daten – die müssten wir sonst mit `pos` und `copy` auseinanderfriemeln, was ich mir an dieser Stelle sparen möchte.

```
procedure TMeMPPlaylist.LoadFromM3U(aFilename: String);
var mylist: TStringlist;
  i: Integer;
  s: String;
begin
  mylist := TStringlist.Create;
  mylist.LoadFromFile(aFilename);
  if (myList.Count > 0) then
  begin
    if (myList[0] = '#EXTM3U') then //Liste ist im EXT-Format
    begin
      i := 1;
      while (i < myList.Count) do
      begin
        // Zuerst kommen ggf. die ExtInf-Daten
        s := myList[i];
        if trim(s) = '' then
          inc(i)
```

```
        else
        begin
          if (copy(s,0,7) = '#EXTINF') then // ExtInf-Zeile überspringen
            inc(i);
          Add(ExpandFilename(myList[i]));
          inc(i);
        end;
      end;
    end
    else
      // Liste ist nicht im EXT-Format - einfach nur Dateinamen
      for i := 0 to myList.Count - 1 do
      begin
        if trim(mylist[i])='' then continue;
        Add(ExpandFilename(myList[i]));
      end;
  end;
  FreeAndNil(myList);
end;
```

Beim Speichern müssen wir weniger Fehler abfangen, daher geht das kürzer:

```
procedure TMeMPPlaylist.SaveAsM3U(aFilename: String);
var myList: tStringlist;
    i:integer;
    aAudiofile: TAudioFile;
begin
  myList := TStringList.Create;
  myList.Add('#EXTM3U');
  for i := 0 to fPlayList.Count - 1 do
  begin
    aAudiofile := fPlaylist[i] as TAudioFile;
    myList.add('#EXTINF:' + IntTostr(aAudiofile.Dauer) + ','
          + aAudioFile.Interpret + ' - ' + aAudioFile.Titel);
    myList.Add(ExtractRelativePath(aFilename, aAudioFile.Pfad ));
  end;
  myList.SaveToFile(afilename);
  FreeAndNil(myList);
end;
```

Wie wir das in unseren Player einbauen, ist hoffentlich klar. Auf die Form einfach noch zwei weiterere Buttons, einen OpenDialog und einen SaveDialog draufpacken und im OnClick-Event der beiden Buttons nach dem Execute des Dialogs die Liste laden oder speichern.

Kapitel 13. Visualisierung

Fast jeder Player hat sie, und auch wir wollen das bei unserem nicht auslassen. Glücklicherweise geht das hier auch ganz einfach. Zunächst durchforsten wir das Archiv, das wir mit der bass.dll heruntergeladen haben, nach dem Ordner mit den Beispielprojekten für Delphi. Dort finden wir einen Ordner SampleVis, in dem sich ein Beispiel-Projekt zur Visualisierung befindet. Wir wollen uns auf die klassische Balkenansicht beschränken und fügen die beiden Units `spectrum_vis.pas` und `CommonTypes.pas` in unser Projekt ein. Die beiden benötigen wir nun für unsere Player-Klasse.

> **Hinweis:** Ja, das bauen wir in die Player-Klasse ein, nicht in die Playlist-Klasse. Das ist etwas, was mit dem Player zu tun hat, nicht mit der ganzen Liste. Da unsere Playlistklasse von der Player-Klasse abgeleitet ist, wird diese damit auch erweitert.

Wir fügen dort zunächst eine weitere Initialisierungsmethode hinzu

```
procedure TMeMPPlayer.InitSpectrum(Width, Height: Integer);
begin
  Spectrum := TSpectrum.Create(Width, Height);
  Spectrum.Mode := 1;
  Spectrum.BackColor := clBtnFace;
  Spectrum.Pen := clActiveCaption;
  Spectrum.Peak := clWindowText;
end;
```

erweitern den Destruktor der Klasse um die Zeile

```
if assigned(Spectrum) then Spectrum.Free;
```

und implementieren zuletzt noch eine Methode, die das Zeichnen übernimmt, bzw. die entsprechende Methode in der gerade kopierten Unit aufruft.

```
procedure TMeMPPlayer.DrawSpectrum(aHandle: THandle);
var FFTFata : TFFTData;
begin
  if BASS_ChannelIsActive(fMainStream) = BASS_ACTIVE_PLAYING then
  begin
    BASS_ChannelGetData(fMainStream, @FFTFata, BASS_DATA_FFT1024);
    Spectrum.Draw (aHandle, FFTFata, 0, 0);
  end;
end;
```

Ich gebe zu, das ist jetzt sehr billig gewesen. Aber jetzt noch auf die FFT-Daten eingehen, und wie man die schnell und einfach darstellt...nö. Irgendwann muss auch mal gut sein. So, wie wir am Anfang das Auslesen der ID3-Tags anderen Klassen überlassen haben, schieben wir jetzt nochmal etwas Arbeit von uns.

Außerdem wollen wir ja jetzt langsam zum Ende kommen, oder? Und das sind wir jetzt auch. Wir müssen nur noch die Visualisierung in den Player einbauen, was mit ein paar Klicks und zwei Zeilen Code erledigt ist.

Kapitel 14. MeMP, Version 0.1

Auf unsere Player-Form setzen wir jetzt noch eine Paintbox, initialisieren die Visualisierung im FormCreate mit

```
MeMPPlaylist.InitSpectrum(SpectrumPaintbox.Width, SpectrumPaintbox.Height);
```

und fügen dem Timer folgende Codezeile hinzu:

```
MeMPPlaylist.DrawSpectrum(SpectrumPaintbox.Canvas.Handle);
```

Und dann bleibt eigentlich nur noch eins zu sagen: **„Ich habe fertig!"**

Zusammenfassung, Ausblick

Der Player ist zwar noch lange nicht wirklich fertig, aber das Gerüst steht. Sicherlich ist es sinnvoll, dem eigenen Player eine eigene, ganz besondere Note zu geben, damit er im Player-Dschungel nicht untergeht. Das kann durch ein ausgefallenes Design geschehen, oder durch Spezialisierung für einen besonderen Einsatzzweck, z.B. Einsatz an einem Rechner, der per Touchscreen bedient wird und an dem sich Partygäste ihre Wunschtitel selbst zusammenstellen können.

Ich wünsche Dir viel Spaß und Erfolg dabei und hoffe, dass ich mit diesem nicht ganz so kurzen Einstieg hilfreich sein konnte.

Daniel Gaußmann,

Dezember 2007