CHAPTER

18

Multimedia Programming with Delphi

IN THIS CHAPTER

Delphi's TMediaPlayer component is proof that good things come in small packages. In the guise of this little component, Delphi encapsulates a great deal of the functionality of the Windows *Media Control Interface* (MCI)—the portion of the Windows API that provides control for multimedia devices.

Delphi makes multimedia programming so easy that the traditional and boring "Hello World" program may be a thing of the past. Why write Hello World to the screen when it's almost as easy to play a sound or video file that offers its greetings?

In this chapter, you learn how to write a simple yet powerful media player, and you even construct a fully functional audio CD player. This chapter explains the uses and nuances of the TMediaPlayer component. Of course, your computer must be equipped with multimedia devices, such as a sound card and CD-ROM, for this chapter to be of real use to you.

Creating a Simple Media Player

The best way to learn is by doing. This application demonstrates how quickly you can create a media player by placing TMediaPlayer, TButton, and TOpenDialog components on a form. This form is shown in Figure 18.1.

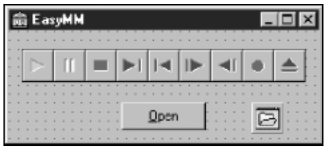


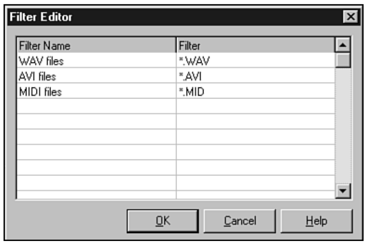
FIGURE 18.1
The EasyMM *Media Player.*

The EasyMM Media Player works like this: After you click Button1, the OpenDialog dialog box appears, and you choose a file from it. The Media Player prepares itself to play the file you chose in OpenDialog. You then can click the Play button on the Media Player to play the file. The following code belongs to the button's OnClick method, and it opens the Media Player with the file you chose:

```
procedure TMainForm.BtnOpenClick(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    MediaPlayer1.Filename := OpenDialog1.Filename;
    MediaPlayer1.Open;
  end;
end;
```

This code executes the `OpenDialog1` dialog box, and if a filename is chosen, `OpenDialog1`'s `FileName` property is copied to `MediaPlayer1`'s `FileName` property. The `MediaPlayer`'s `Open` method is then called to prepare it to play the file.

You might also want to limit the files to browse through with the `OpenDialog` dialog box to only multimedia files. `TMediaPlayer` supports a whole gaggle of multimedia device types, but for now, you'll just browse WAV, AVI, and MIDI files. This capability exists in the `TOpenDialog` component, and you take advantage of it by selecting `OpenDialog1` in the Object Inspector, choosing the `Mask` property, and clicking the ellipsis to the right of this item to invoke the Filter Editor. Fill in the `.WAV`, `.AVI`, and `.MID` extensions, as shown in Figure 18.2.



Filter Name	Filter
WAV files	*.WAV
AVI files	*.AVI
MIDI files	*.MID

FIGURE 18.2
The Filter Editor.

The project is saved as `EasyMM.dpr` and the main unit as `Main.pas`. The Media Player is now ready to run. Run the program, and try it out using one of the multimedia files on your hard disk. Other people might have convinced you—or perhaps you had convinced yourself—that multimedia programming is difficult, but now you have firsthand proof that this just isn't true.

Using WAV Files in Your Applications

WAV files (pronounced *wave*, which is short for *waveform*) are the standard file format for sharing audio in Windows. As the name implies, WAV files store sounds in a binary format that resembles a mathematical wave. The great thing about WAV files is that they have gained industry acceptance, and you can find them everywhere. The bad thing about WAV files is that they tend to be bulky, and just a few of those Homer Simpson WAV files can take up a hefty chunk of hard disk space.

The `TMediaPlayer` component enables you to easily integrate WAV sounds into your applications. As just illustrated, playing WAV files in your program is no sweat—just feed a `TMediaPlayer` component a filename, open it, and play it. A little audio capability can be just the thing your applications need to go from neat to way cool.

If playing WAV files is all you want to do, you might not need the overhead of a `TMediaPlayer` component. Instead, you can use the `PlaySound()` API function found in the `MMSystem` unit. `PlaySound()` is defined this way:

```
function PlaySound(pszSound: PChar; hmod: HMODULE;
  fdwSound: DWORD): BOOL; stdcall;
```

`PlaySound()` has the capability to play a WAV sound from a file, from memory, or from a resource file linked into the application. `PlaySound()` accepts three parameters:

- The first parameter, `pszSound`, is a `PChar` variable that represents a filename, alias name, resource name, Registry entry, entry from the `[sounds]` section of your `WIN.INI` file, or pointer to a WAV sound located somewhere in memory.
- The second parameter, `hmod`, represents the handle of the executable file that contains the resource to be loaded. This parameter must be zero unless `snd_Resource` is specified in the `fdwSound` parameter.
- The third parameter, `fdwSound`, contains flags that describe how the sound should be played. These flags can contain a combination of any of the following values:

Flag	*Description*
`SND_APPLICATION`	The sound is played using an application-specific association.
`SND_ALIAS`	The `pszSound` parameter is a system-event alias in the Registry or the `WIN.INI` file. Don't use this flag with either `SND_FILENAME` or `SND_RESOURCE`, because they're mutually exclusive.
`SND_ALIAS_ID`	The `pszSound` parameter is a predefined sound identifier.
`SND_FILENAME`	The `pszSound` parameter is a filename.
`SND_NOWAIT`	This flag indicates that if the driver is busy, it returns immediately without playing the sound.
`SND_PURGE`	All sounds are stopped for the calling task. If `pszSound` is not zero, all instances of the specified sound are stopped. If `pszSound` is zero, all sounds invoked by the current task are stopped. You must also specify the proper instance handle to stop `SND_RESOURCE` events.
`SND_RESOURCE`	The `pszSound` parameter is a resource identifier. When you're using this flag, the `hmod` parameter must contain the instance that contains the specified resource.
`SND_ASYNC`	Plays the sound asynchronously and returns the function almost immediately. This achieves a background music effect.
`SND_LOOP`	Plays the sound over and over until you make it stop or you go insane. `SND_ASYNC` also must be specified when using this flag.

`SND_MEMORY`	Plays the WAV sound in the memory area pointed to by the `pszSound` parameter.
`SND_NODEFAULT`	If the sound can't be found, `PlaySound()` returns immediately without playing the default sound, as specified in the Registry.
`SND_NOSTOP`	Plays the sound only if it isn't already playing. `PlaySound()` returns `True` if the sound is played and `False` if the sound is not played. If this flag is not specified, Win32 will stop any currently playing sound before attempting to play the sound specified in `pszSound`.
`SND_SYNC`	Plays the sound synchronously and doesn't return from the function until the sound finishes playing.

TIP

To terminate a WAV sound currently playing asynchronously, call `PlaySound()` and pass `Nil` or zero for all parameters, as follows:

```
PlaySound(Nil, 0, 0);  // stop currently playing WAV
```

To terminate even non-waveform sounds for a given task, add the `snd_Purge` flag:

```
PlaySound(Nil, 0, snd_Purge);  // stop all currently playing sounds
```

NOTE

The Win32 API still supports the `sndPlaySound()` function, which was a part of the Windows 3.*x* API. This function is only supported for backward compatibility, however, and it might not be available in future implementations of the Win32 API. Use the Win32 `PlaySound()` function rather than `sndPlaySound()` for future compatibility.

Playing Video

AVI (short for *audio-video interleave)* is one of the most common file formats used to exchange audio and video information simultaneously. In fact, you'll find a couple of AVI files in the `\Runimage\Delphi50\Demos\Coolstuf` directory of the CD-ROM that contains your copy of Delphi 5.

You can use the simple multimedia player program you wrote earlier in this chapter to display AVI files. Simply select an AVI file when `OpenDialog1` is invoked and click the Play button. Note that the AVI file plays in its own window.

Showing the First Frame

You might want to display the first frame of an AVI file in a window before you actually play the file. This achieves a sort of freeze-frame effect. To do this after opening the `TMediaPlayer` component, just set the `Frames` property of `TMediaPlayer` to `1` and then call the `Step()` method. The `Frames` property tells `TMediaPlayer` how many frames to move when `Step()` and `Back()` methods are called. `Step()` advances the `TMediaPlayer` frames and displays the current frame. This is the code:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    with MediaPlayer1 do
    begin
      Filename := OpenDialog1.Filename;
      Open;
      Frames := 1;
      Step;
      Notify := True;
    end;
end;
```

Using the Display Property

You can assign a value to `TMediaPlayer`'s `Display` property to cause the AVI file to play to a specific window, instead of creating its own window. To do this, you add a `TPanel` component to your Media Player, as shown in Figure 18.3. After adding the panel, you can save the project in a new directory as `DDGMPlay.dpr`.

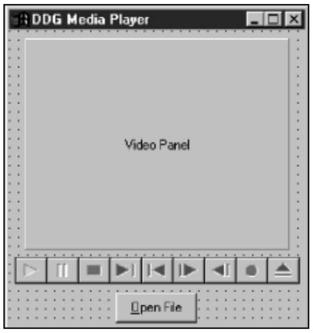


FIGURE 18.3
The `DDGMPlay` *main window.*

Click the drop-down arrow button for `MediaPlayer1`'s `Display` property and notice that all the components in this project appear in the list box. Set the value of the `Display` property to `Panel1`.

Now notice that when you run the program and select and play an AVI file, the AVI file output appears in the panel. Also notice that the AVI file doesn't take up the whole area of the panel; the AVI file has a certain default size programmed into it.

Using the DisplayRect Property

`DisplayRect` is a property of type `TRect` that determines the size of the AVI file output window. You can use the `DisplayRect` property to cause your AVI file's output to stretch or shrink to a certain size. If you want the AVI file to take up the whole area of `Panel1`, for example, you can assign `DisplayRect` to the size of the panel:

```
MediaPlayer1.DisplayRect := Rect(0, 0, Panel1.Width, Panel1.Height);
```

You can add this line of code to the `OnClick` handler for `Button1`, like this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then begin
    MediaPlayer1.Filename := OpenDialog1.Filename;
    MediaPlayer1.Open;
    MediaPlayer1.DisplayRect := Rect(0, 0, Panel1.Width, Panel1.Height);
  end;
end;
```

CAUTION

You can set the `DisplayRect` property only after the `TMediaPlayer`'s `Open()` method is called.

Understanding TMediaPlayer Events

`TMediaPlayer` has two unique events: `OnPostClick` and `OnNotify`.

The `OnPostClick` event is very similar to `OnClick`, but `OnClick` occurs as soon as the component is clicked, and `OnPostClick` executes only after some action occurs that was caused by a click. If you click the Play button on `TMediaPlayer` at runtime, for example, an `OnClick` event is generated, but an `OnPostClick` event is generated only after the media device is done playing.

The `OnNotify` event is a little more interesting. The `OnNotify` event executes whenever the `TMediaPlayer` completes a media-control method (such as `Back`, `Close`, `Eject`, `Next`, `Open`, `Pause`, `PauseOnly`, `Play`, `Previous`, `Resume`, `Rewind`, `StartRecording`, `Step`, or `Stop`) and only when `TMediaPlayer`'s `Notify` property is set to `True`. To illustrate `OnNotify`, add a handler for this event to the DDGMPlay project. In the event handler method, you cause a message dialog box to appear after a command executes:

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  MessageDlg('Media control method executed', mtInformation, [mbOk], 0);
end;
```

Don’t forget to also set the `Notify` property to `True` in `Button1`’s `OnClick` handler after opening the Media Player:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    with MediaPlayer1 do
    begin
      Filename := OpenDialog1.Filename;
      Open;
      DisplayRect := Rect(0, 0, Panel1.Width, Panel1.Height);
      Notify := True;
    end;
end;
```

TIP

Notice that you moved the code dealing with `MediaPlayer1` into a `with..do` construct. As you learned in earlier chapters, this construct offers advantages in code clarity and performance over simply qualifying each property and method name.

Viewing the Source Code for DDGMPlay

By now, you should know the basics of how to play WAV and AVI files. Listings 18.1 and 18.2 show the complete source code for the `DDGMPlay` project.

LISTING 18.1 The Source Code for DDGMPlay.dpr

```
program DDGMPlay;

uses
  Forms,
  Main in 'MAIN.PAS' {MainForm};

{$R *.RES}

begin
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

LISTING 18.2 The Source Code for Main.pas

```
unit Main;

interface

uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, MPlayer, ExtCtrls;

type
  TMainForm = class(TForm)
    MediaPlayer1: TMediaPlayer;
    OpenDialog1: TOpenDialog;
    Button1: TButton;
    Panel1: TPanel;
    procedure Button1Click(Sender: TObject);
    procedure MediaPlayer1Notify(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
    with MediaPlayer1 do
    begin
      Filename := OpenDialog1.Filename;
      Open;
      DisplayRect := Rect(0, 0, Panel1.Width, Panel1.Height);
      Notify := True;
    end;
end;

procedure TMainForm.MediaPlayer1Notify(Sender: TObject);
begin
  MessageDlg('Media control method executed', mtInformation, [mbOk], 0);
```

continues

LISTING 18.2 Continued

```
end;

end.
```

Device Support

`TMediaPlayer` supports the vast array of media devices supported by MCI. The type of device that `TMediaPlayer` controls is determined by its `DeviceType` property. Table 18.1 describes the different values of the `DeviceType` property.

TABLE 18.1 Values of `TMediaPlayer`'s `DeviceType` Property

DeviceType Value	*Media Device*
`dtAutoSelect`	The `TMediaPlayer` automatically should select the correct device type based on the filename to be played.
`dtAVIVideo`	AVI file. These files have the `.AVI` extension and contain both sound and full-motion video.
`dtCDAudio`	An audio CD played from your computer's CD-ROM drive.
`dtDAT`	A digital audio tape (DAT) player connected to your PC.
`dtDigitalVideo`	A digital video device, such as a digital video camera.
`dtMMMovie`	A multimedia movie format.
`dtOther`	An unspecified multimedia format.
`dtOverlay`	A video overlay device.
`dtScanner`	A scanner connected to your PC.
`dtSequencer`	A sequencer device capable of playing MIDI files. MIDI files typically end in a `.MID` or `.RMI` extension.
`dtVCR`	A video cassette recorder (VCR) connected to your PC.
`dtVideodisc`	A video disc player connected to your PC.
`dtWaveAudio`	A WAV audio file. These files end in the `.WAV` extension.

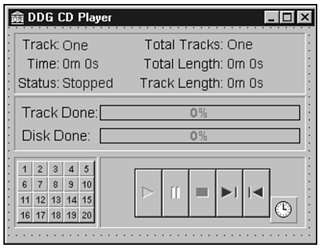
Although you can see that `TMediaPlayer` supports many formats, this chapter focuses primarily on the WAV, AVI, and CD Audio formats because those are the most common under Windows.

NOTE

The `TMediaPlayer` component is a `TWinControl` descendant, which means it can be easily encapsulated as an ActiveX control through the Delphi 5 wizards. One possible benefit of doing this is the ability to embed a Media Player in a Web page to extend your pages with custom multimedia. Additionally, with a few lines of JavaScript or VBScript, you could provide a CD player for everyone on the Internet or your intranet running a Windows browser.

Creating an Audio CD Player

You'll learn about the finer points of the `TMediaPlayer` component by creating a full-featured audio CD player. Figure 18.4 shows the main form for this application, which is called `CDPlayer.dpr`. The main unit for this form is called `CDMain.pas`.



Track: One Total Tracks: One
Time: 0m 0s Total Length: 0m 0s
Status: Stopped Track Length: 0m 0s
Track Done: 0%
Disk Done: 0%

FIGURE 18.4
The audio CD player's main form.

Table 18.2 shows the important properties to be set for the components contained on the CD player's main form.

TABLE 18.2 Important Properties for the CD Player's Components

Component	*Property*	*Value*
`mpCDPlayer`	`DeviceType`	`dtAudioCD`
`sbTrack1-sbTrack20`	`Caption`	`'1' - '20'`
`sbTrack1-sbTrack20`	`Tag`	`1 - 20`

Displaying a Splash Screen

When the CD player is run, it takes a couple seconds for it to load, and it might take several more seconds for the `TMediaPlayer` component to initialize after calling its `Open()` method. This delay from the time the user clicks the icon in Explorer to the time he or she actually sees the program often gives the user an "is my program gonna start, or isn't it?" feeling. This delay is caused by the time Windows takes to load its multimedia subsystem, which occurs when TMediaPlayer is opened. To avoid this problem, you can give the CD player program a splash screen that displays as the program starts. The splash screen tells users that, yes, the program will eventually start—it's just taking a moment to load, so enjoy this little screen in the meantime.

The first step in creating a splash screen is to create a form that you want to use as the splash screen. Generally, you want this form to contain a panel but not a border or title bar; this gives it a 3D, floating-panel appearance. On the panel, place one or more TLabel components and perhaps a TImage component that displays a bitmap or icon.

The splash screen form for the CD player is shown in Figure 18.5, and the unit, Splash.pas, is shown in Listing 18.3.

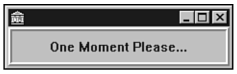


FIGURE 18.5
The CD player's splash screen form.

LISTING 18.3 The Source Code for SPLASH.PAS

```
unit Splash;
interface

uses Windows, Classes, Graphics, Forms, Controls, StdCtrls,
  ExtCtrls;

type
  TSplashScreen = class(TForm)
    StatusPanel: TPanel;
  end;

var
  SplashScreen: TSplashScreen;

implementation

{$R *.DFM}

begin
  { Since the splash screen is displayed before the main screen is created,
    it must be created before the main screen. }
  SplashScreen := TSplashScreen.Create(Application);
  SplashScreen.Show;
  SplashScreen.Update;
end.
```

Unlike a normal form, the splash screen is created and shown in the `initialization` section of its unit. Because the `initialization` section for all units is executed before the main program block in the DPR file, this form is displayed before the main portion of the program runs.

CAUTION

Do *not* use `Application.CreateForm()` to create your splash screen form instance. The first time `Application.CreateForm()` is called in an application, Delphi makes that form the main application form. It would be a "bad thing" to make your splash screen the main form.

Beginning the CD Player

Create an event handler for the form's `OnCreate` method. In this method, you open and initialize the CD player program. First, call `CDPlayer`'s `Open()` method. `Open()` checks to make sure that the system is capable of playing audio CDs and then initializes the device. If `Open()` fails, it raises an exception of type `EMCIDeviceError`. In the event of an exception opening the device, you should terminate the application. Here's the code:

```
try
  mpCDPlayer.Open; { Open the CD Player device. }
except
  { If an error occurred, the system may be incapable of playing CDs. }
  on EMCIDeviceError do
  begin
    MessageDlg('Error Initializing CD Player.  Program will now exit.',
               mtError, [mbOk], 0);
    Application.Terminate;   { bail out }
  end;
end;
```

NOTE

The preferred way to end a Delphi application is by calling the main form's `Close()` method or by calling `Application.Terminate`.

After opening `CDPlayer`, you should set its `EnabledButtons` property to ensure that the proper buttons are enabled for the device. Which buttons to enable, however, depends on the current state of the CD device. If a CD is already playing when you call `Open()`, for example, you obviously don't want to enable the Play button. To perform a check on the current status of the CD device, you can inspect `CDPlayer`'s `Mode` property. The `Mode` property, which has all its possible values laid out nicely in the online help, provides information on whether a CD device is currently playing, stopped, paused, seeking, and so on. In this case, your concern is only whether the device is stopped, paused, or playing. The following code enables the proper buttons:

```
case mpCDPlayer.Mode of
  mpPlaying: mpCDPlayer.EnabledButtons := [btPause, btStop, btNext, btPrev];
  mpStopped,        { show default buttons if stopped }
  mpPaused : mpCDPlayer.EnabledButtons := [btPlay, btNext, btPrev];
end;
```

The following is the completed source code for the `TMainForm.FormCreate()` method. Notice that you make calls to several methods after successfully opening `CDPlayer`. The purpose of these methods is to update various aspects of the CD player application, such as the number of tracks on the current CD and the current track position. (These methods are described in more detail later in this chapter.) Here's the code:

```
procedure TMainForm.FormCreate(Sender: TObject);
{ This method is called when the form is created. It opens and initializes the
  player }
begin
  try
    mpCDPlayer.Open;    // Open the CD Player device.
    { If a CD is already playing at startup, show playing status. }
    if mpCDPlayer.Mode = mpPlaying then
      LblStatus.Caption := 'Playing';
    GetCDTotals;      // Show total time and tracks on current CD
    ShowTrackNumber;  // Show current track
    ShowTrackTime;    // Show the minutes and seconds for the current track
    ShowCurrentTime;  // Show the current position of the CD
    ShowPlayerStatus; // Update the CD Player's status
  except
    { If an error occurred, the system may be incapable of playing CDs. }
    on EMCIDeviceError do
    begin
      MessageDlg('Error Initializing CD Player.  Program will now exit.',
                 mtError, [mbOk], 0);
      Application.Terminate;
    end;
  end;
  { Check the current mode of the CD-ROM and enable the appropriate buttons. }
  case mpCDPlayer.Mode of
```

```
    mpPlaying: mpCDPlayer.EnabledButtons := PlayButtons;
    mpStopped, mpPaused: mpCDPlayer.EnabledButtons := StopButtons;
  end;
  SplashScreen.Release;  // Close and free the splash screen
end;
```

Notice that the last line of code in this method closes the splash screen form. The `OnCreate` event of the main form is generally the best place to do this.

Updating the CD Player Information

As the CD device plays, you can keep the information on `CDPlayerForm` up-to-date by using a `TTimer` component. Every time a timer event occurs, you can call the necessary updating methods, as shown in the form's `OnCreate` method, to ensure that the display stays current. Double-click `Timer1` to generate a method skeleton for its `OnTimer` event. Here's the source code you use for this event:

```
procedure TMainForm.tmUpdateTimerTimer(Sender: TObject);
{ This method is the heart of the CD Player.  It updates all information at
  every timer interval. }
begin
  if mpCDPlayer.EnabledButtons = PlayButtons then
  begin
    mpCDPlayer.TimeFormat := tfMSF;
    ggDiskDone.Progress := (mci_msf_minute(mpCDPlayer.Position) * 60 +
                           mci_msf_second(mpCDPlayer.Position));
    mpCDPlayer.TimeFormat := tfTMSF;
    ShowTrackNumber; // Show track number the CD player is currently on
    ShowTrackTime;   // Show total time for the current track
    ShowCurrentTime; // Show elapsed time for the current track
  end;
end;
```

Notice that, in addition to calling the various updating methods, this method also updates the `DiskDoneGauge` control for the amount of time elapsed on the current CD. To get the elapsed time, the method changes `CDPlayer`'s `TimeFormat` property to `tfMSF` and gets the minute and second value from the `Position` property by using the `mci_msf_Minute()` and `mci_msf_Second()` functions. This merits a bit more explanation.

TimeFormat

The `TimeFormat` property of a `TMediaPlayer` component determines how the values of the `StartPos`, `Length`, `Position`, `Start`, and `EndPos` properties should be interpreted. Table 18.3 lists the possible values for `TimeFormat`. These values represent information packed into a `Longint` type variable.

TABLE 18.3 Values for the `TMediaPlayer.TimeFormat` Property

Value	*Time Storage Format*
`tfBytes`	Number of bytes
`tfFrames`	Frames
`tfHMS`	Hours, minutes, and seconds
`tfMilliseconds`	Time in milliseconds
`tfMSF`	Minutes, seconds, and frames
`tfSamples`	Number of samples
`tfSMPTE24`	Hours, minutes, seconds, and frames based on 24 frames per second
`tfSMPTE25`	Hours, minutes, seconds, and frames based on 25 frames per second
`tfSMPTE30`	Hours, minutes, seconds, and frames based on 30 frames per second
`tfSMPTE30Drop`	Hours, minutes, seconds, and frames based on 30 drop frames per second
`tfTMSF`	Tracks, minutes, seconds, and frames

Time-Conversion Routines

The Windows API provides routines to retrieve the time information from the different packed formats shown in Table 18.4. *Packed format* means that multiple data values are packed (encoded) into one `Longint` value. These functions are located in `MMSystem.dll`, so be sure to have `MMSystem` in your `uses` clause when using them.

TABLE 18.4 Functions to Unpack Multimedia Time Formats

Function	*Works With*	*Returns*
`mci_HMS_Hour()`	`tfHMS`	Hours
`mci_HMS_Minute()`	`tfHMS`	Minutes
`mci_HMS_Second()`	`tfHMS`	Seconds
`mci_MSF_Frame()`	`tfMSF`	Frames
`mci_MSF_Minute()`	`tfMSF`	Minutes
`mci_MSF_Second()`	`tfMSF`	Seconds
`mci_TMSF_Frame()`	`tfTMSF`	Frames
`mci_TMSF_Minute()`	`tfTMSF`	Minutes
`mci_TMSF_Second()`	`tfTMSF`	Seconds
`mci_TMSF_Track()`	`tfTMSF`	Tracks

Methods for Updating the CD Player

As you learned earlier in this chapter, you use several methods to help keep the information displayed by the CD player up-to-date. The primary purpose of each of these methods is to update the labels in the top portion of the CD player form and to update the gauges in the middle portion of that form.

GetCDTotals()

The purpose of the `GetCDTotals()` method, shown in the following code, is to retrieve the length and total number of tracks on the current CD. This information is then used to update several labels and `DiskDoneGauge`. This code also calls the `AdjustSpeedButtons()` method, which enables the same number of speedbuttons as tracks. Notice that this method also makes use of the `TimeFormat` and time-conversion routines discussed earlier:

```
procedure TMainForm.GetCDTotals;
{ This method gets the total time and tracks of the CD and displays them. }
var
  TimeValue: longint;
begin
  mpCDPlayer.TimeFormat := tfTMSF;                  // set time format
  TimeValue := mpCDPlayer.Length;                   // get CD length
  TotalTracks := mci_Tmsf_Track(mpCDPlayer.Tracks); // get total tracks
  TotalLengthM := mci_msf_Minute(TimeValue);     // get total length in mins
  TotalLengthS := mci_msf_Second(TimeValue);     // get total length in secs
  { set caption of Total Tracks label }
  LblTotTrk.Caption := TrackNumToString(TotalTracks);
  { set caption of Total Time label }
  LblTotalLen.Caption := Format(MSFormatStr, [TotalLengthM, TotalLengthS]);
  { initialize gauge }
  ggDiskDone.MaxValue := (TotalLengthM * 60) + TotalLengthS;
  { enable the correct number of speed buttons }
  AdjustSpeedButtons;
end;
```

ShowCurrentTime()

The `ShowCurrentTime()` method is shown in the following code. This method is designed to obtain the elapsed minutes and seconds for the currently playing track as well as to update the necessary controls. Here, you also use the time-conversion routines provided by `MMSystem`:

```
procedure TMainForm.ShowCurrentTime;
{ This method displays the current time of the current track }
begin
  { Minutes for this track }
  m := mci_Tmsf_Minute(mpCDPlayer.Position);
  { Seconds for this track }
  s := mci_Tmsf_Second(mpCDPlayer.Position);
  { update track time label }
  LblTrackTime.Caption := Format(MSFormatStr, [m, s]);
  { update track gauge }
  ggTrackDone.Progress := (60 * m) + s;
end;
```

ShowTrackTime()

The `ShowTrackTime()` method, shown in the following code, obtains the total length of the current track in minutes and seconds, and it updates a label control. Again, you make use of the time-conversion routines. Also notice that you check to make sure that the track isn't the same as when this function was last called. This comparison ensures that you don't make unnecessary function calls or repaint components unnecessarily. Here's the code:

```
procedure TMainForm.ShowTrackTime;
{ This method changes the track time to display the total length of the
  currently selected track. }
var
  Min, Sec: Byte;
  Len: Longint;
begin
  { Don't update the information if player is still on the same track }
  if CurrentTrack <> OldTrack then
  begin
    Len := mpCDPlayer.TrackLength[mci_Tmsf_Track(mpCDPlayer.Position)];
    Min := mci_msf_Minute(Len);
    Sec := mci_msf_Second(Len);
    ggTrackDone.MaxValue := (60 * Min) + Sec;
    LblTrackLen.Caption := Format(MSFormatStr, [m, s]);
  end;
  OldTrack := CurrentTrack;
end;
```

CD Player Source

You've now seen all aspects of the CD player as they relate to multimedia. Listings 18.4 and 18.5 show the complete source code for the `CDPlayer.dpr` and `CDMain.pas` modules. The `CDMain` unit also shows some of the techniques you use to manipulate the speedbuttons using their `Tag` properties as well as other techniques for updating the controls.

LISTING 18.4 The Source Code for CDPlayer.dpr

```
program CDPlayer;

uses
  Forms,
  Splash in 'Splash.pas' {SplashScreen},
  CDMain in 'CDMain.pas' {MainForm};

begin
  Application.CreateForm(TMainForm, MainForm);
  Application.Run;
end.
```

LISTING 18.5 The Source Code for CDMain.pas

```
unit CDMain;

interface

uses
  SysUtils, Windows, Classes, Graphics, Forms, Controls, MPlayer, StdCtrls,
  Menus, MMSystem, Messages, Buttons, Dialogs, ExtCtrls, Splash, Gauges;

type
  TMainForm = class(TForm)
    tmUpdateTimer: TTimer;
    MainScreenPanel: TPanel;
    LblStatus: TLabel;
    Label2: TLabel;
    LblCurTrk: TLabel;
    Label4: TLabel;
    LblTrackTime: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    LblTotTrk: TLabel;
    LblTotalLen: TLabel;
    Label12: TLabel;
    LblTrackLen: TLabel;
    Label15: TLabel;
    CDInfo: TPanel;
    SBPanel: TPanel;
    Panel1: TPanel;
    mpCDPlayer: TMediaPlayer;
    sbTrack1: TSpeedButton;
    sbTrack2: TSpeedButton;
    sbTrack3: TSpeedButton;
    sbTrack4: TSpeedButton;
    sbTrack5: TSpeedButton;
    sbTrack6: TSpeedButton;
    sbTrack7: TSpeedButton;
    sbTrack8: TSpeedButton;
    sbTrack9: TSpeedButton;
```

continues

LISTING 18.5 Continued

```
    sbTrack10: TSpeedButton;
    sbTrack11: TSpeedButton;
    sbTrack12: TSpeedButton;
    sbTrack13: TSpeedButton;
    sbTrack14: TSpeedButton;
    sbTrack15: TSpeedButton;
    sbTrack16: TSpeedButton;
    sbTrack17: TSpeedButton;
    sbTrack18: TSpeedButton;
    sbTrack19: TSpeedButton;
    sbTrack20: TSpeedButton;
    ggTrackDone: TGauge;
    ggDiskDone: TGauge;
    Label1: TLabel;
    Label3: TLabel;
    procedure tmUpdateTimerTimer(Sender: TObject);
    procedure mpCDPlayerPostClick(Sender: TObject; Button: TMPBtnType);
    procedure FormCreate(Sender: TObject);
    procedure sbTrack1Click(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
    OldTrack, CurrentTrack: Byte;
    m, s: Byte;
    TotalTracks: Byte;
    TotalLengthM: Byte;
    TotalLengthS: Byte;
    procedure GetCDTotals;
    procedure ShowTrackNumber;
    procedure ShowTrackTime;
    procedure ShowCurrentTime;
    procedure ShowPlayerStatus;
    procedure AdjustSpeedButtons;
    procedure HighlightTrackButton;
    function TrackNumToString(InNum: Byte): String;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

const
```

```
  { Array of strings representing numbers from one to twenty: }
  NumStrings: array[1..20] of String[10] =
      ('One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine',
       'Ten', 'Eleven', 'Twelve', 'Thirteen', 'Fourteen', 'Fifteen', 'Sixteen',
       'Seventeen', 'Eighteen', 'Nineteen', 'Twenty');
  MSFormatStr = '%dm %ds';
  PlayButtons: TButtonSet = [btPause, btStop, btNext, btPrev];
  StopButtons: TButtonSet = [btPlay, btNext, btPrev];

function TMainForm.TrackNumToString(InNum: Byte): String;
{ This function returns a string corresponding to a integer between 1 and 20.
  If the number is greater than 20, then the integer is returned as a string. }
begin
  if (InNum > High(NumStrings)) or (InNum < Low(NumStrings)) then
    Result := IntToStr(InNum)    { if not in array, then just return number }
  else
    Result := NumStrings[InNum]; { return the string from NumStrings array }
end;

procedure TMainForm.AdjustSpeedButtons;
{ This method enables the proper number of speed buttons }
var
  i: integer;
begin
  { iterate through form's Components array... }
  for i := 0 to SBPanel.ControlCount - 1 do
    if SBPanel.Controls[i] is TSpeedButton then    // is it a speed button?
      { disable buttons higher than number of tracks on CD }
      with TSpeedButton(SBPanel.Controls[i]) do Enabled := Tag <= TotalTracks;
end;


procedure TMainForm.GetCDTotals;
{ This method gets the total time and tracks of the CD and displays them. }
var
  TimeValue: longint;
begin
  mpCDPlayer.TimeFormat := tfTMSF;                  // set time format
  TimeValue := mpCDPlayer.Length;                   // get CD length
  TotalTracks := mci_Tmsf_Track(mpCDPlayer.Tracks); // get total tracks
  TotalLengthM := mci_msf_Minute(TimeValue);     // get total length in mins
  TotalLengthS := mci_msf_Second(TimeValue);     // get total length in secs
  { set caption of Total Tracks label }
  LblTotTrk.Caption := TrackNumToString(TotalTracks);
  { set caption of Total Time label }
  LblTotalLen.Caption := Format(MSFormatStr, [TotalLengthM, TotalLengthS]);
```

continues

LISTING 18.5 Continued

```
  { intitialize gauge }
  ggDiskDone.MaxValue := (TotalLengthM * 60) + TotalLengthS;
  { enable the correct number of speed buttons }
  AdjustSpeedButtons;
end;

procedure TMainForm.ShowPlayerStatus;
{ This method displays the status of the CD Player and the CD that
  is currently being played. }
begin
  if mpCDPlayer.EnabledButtons = PlayButtons then
    with LblStatus do
    begin
      case mpCDPlayer.Mode of
        mpNotReady: Caption := 'Not Ready';
        mpStopped:  Caption := 'Stopped';
        mpSeeking:  Caption := 'Seeking';
        mpPaused:   Caption := 'Paused';
        mpPlaying:  Caption := 'Playing';
      end;
    end
  { If these buttons are displayed the CD Player must be stopped... }
  else if mpCDPlayer.EnabledButtons = StopButtons then
    LblStatus.Caption := 'Stopped';
end;

procedure TMainForm.ShowCurrentTime;
{ This method displays the current time of the current track }
begin
  { Minutes for this track }
  m := mci_Tmsf_Minute(mpCDPlayer.Position);
  { Seconds for this track }
  s := mci_Tmsf_Second(mpCDPlayer.Position);
  { update track time label }
  LblTrackTime.Caption := Format(MSFormatStr, [m, s]);
  { update track gauge }
  ggTrackDone.Progress := (60 * m) + s;
end;

procedure TMainForm.ShowTrackTime;
{ This method changes the track time to display the total length of the
  currently selected track. }
var
  Min, Sec: Byte;
  Len: Longint;
```

```
begin
  { Don't update the information if player is still on the same track }
  if CurrentTrack <> OldTrack then
  begin
    Len := mpCDPlayer.TrackLength[mci_Tmsf_Track(mpCDPlayer.Position)];
    Min := mci_msf_Minute(Len);
    Sec := mci_msf_Second(Len);
    ggTrackDone.MaxValue := (60 * Min) + Sec;
    LblTrackLen.Caption := Format(MSFormatStr, [m, s]);
  end;
  OldTrack := CurrentTrack;
end;

procedure TMainForm.HighlightTrackButton;
{ This procedure changes the color of the speedbutton font for the current
  track to red, while changing other speedbuttons to navy blue. }
var
  i: longint;
begin
  { iterate through form's components }
  for i := 0 to ComponentCount - 1 do
    { is it a speedbutton? }
    if Components[i] is TSpeedButton then
      if TSpeedButton(Components[i]).Tag = CurrentTrack then
        { turn red if current track }
        TSpeedButton(Components[i]).Font.Color := clRed
      else
        { turn blue if not current track }
        TSpeedButton(Components[i]).Font.Color := clNavy;
end;

procedure TMainForm.ShowTrackNumber;
{ This method displays the currently playing track number. }
var
  t: byte;
begin
  t := mci_Tmsf_Track(mpCDPlayer.Position);     // get current track
  CurrentTrack := t;                           // set instance variable
  LblCurTrk.Caption := TrackNumToString(t); // set Curr Track label caption
  HighlightTrackButton;                        // Highlight current speedbutton
end;

procedure TMainForm.tmUpdateTimerTimer(Sender: TObject);
{ This method is the heart of the CD Player.  It updates all information at
  every timer interval. }
begin
```

continues

LISTING 18.5 Continued

```
  if mpCDPlayer.EnabledButtons = PlayButtons then
  begin
    mpCDPlayer.TimeFormat := tfMSF;
    ggDiskDone.Progress := (mci_msf_minute(mpCDPlayer.Position) * 60 +
                           mci_msf_second(mpCDPlayer.Position));
    mpCDPlayer.TimeFormat := tfTMSF;
    ShowTrackNumber; // Show track number the CD player is currently on
    ShowTrackTime;   // Show total time for the current track
    ShowCurrentTime; // Show elapsed time for the current track
  end;
end;

procedure TMainForm.mpCDPlayerPostClick(Sender: TObject;
  Button: TMPBtnType);
{ This method displays the correct CD Player buttons when one of the buttons
  are clicked. }
begin
  Case Button of
    btPlay:
      begin
        mpCDPlayer.EnabledButtons := PlayButtons;
        LblStatus.Caption := 'Playing';
      end;
    btPause:
      begin
        mpCDPlayer.EnabledButtons := StopButtons;
        LblStatus.Caption := 'Paused';
      end;
    btStop:
      begin
        mpCDPlayer.Rewind;
        mpCDPlayer.EnabledButtons := StopButtons;
        LblCurTrk.Caption := 'One';
        LblTrackTime.Caption := '0m 0s';
        ggTrackDone.Progress := 0;
        ggDiskDone.Progress := 0;
        LblStatus.Caption := 'Stopped';
      end;
    btPrev, btNext:
      begin
        mpCDPlayer.Play;
        mpCDPlayer.EnabledButtons := PlayButtons;
        LblStatus.Caption := 'Playing';
      end;
  end;
```

```
end;

procedure TMainForm.FormCreate(Sender: TObject);
{ This method is called when the form is created. It opens and initializes the
  player }
begin
  try
    mpCDPlayer.Open;    // Open the CD Player device.
    { If a CD is already playing at startup, show playing status. }
    if mpCDPlayer.Mode = mpPlaying then
      LblStatus.Caption := 'Playing';
    GetCDTotals;      // Show total time and tracks on current CD
    ShowTrackNumber;  // Show current track
    ShowTrackTime;    // Show the minutes and seconds for the current track
    ShowCurrentTime;  // Show the current position of the CD
    ShowPlayerStatus; // Update the CD Player's status
  except
    { If an error occurred, the system may be incapable of playing CDs. }
    on EMCIDeviceError do
    begin
      MessageDlg('Error Initializing CD Player.  Program will now exit.',
                 mtError, [mbOk], 0);
      Application.Terminate;
    end;
  end;
  { Check the current mode of the CD-ROM and enable the appropriate buttons. }
  case mpCDPlayer.Mode of
    mpPlaying: mpCDPlayer.EnabledButtons := PlayButtons;
    mpStopped, mpPaused: mpCDPlayer.EnabledButtons := StopButtons;
  end;
  SplashScreen.Release;  // Close and free the splash screen
end;

procedure TMainForm.sbTrack1Click(Sender: TObject);
{ This method sets the current track when the user presses one of the track
  speed buttons.  This method works with all 20 speed buttons, so by looking at
  the 'Sender' it can tell which button was pressed by the button's tag. }
begin
  mpCDPlayer.Stop;
  { Set the start position on the CD to the start of the newly selected track }
  Track := (Sender as TSpeedButton).Tag;
  mpCDPlayer.StartPos := mpCDPlayer.TrackPosition[Track];
  { Start playing CD at new position }
  mpCDPlayer.Play;
  mpCDPlayer.EnabledButtons := PlayButtons;
  LblStatus.Caption := 'Playing';
```

continues

LISTING 18.5 Continued

```
end;

procedure TMainForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  mpCDPlayer.Close;
end;

end.
```

Summary

That about wraps up the basic concepts of Delphi's `TMediaPlayer` component. This chapter demonstrates the power and simplicity of this component through several examples. In particular, you learned about the common multimedia formats of WAV audio, AVI audio/video, and CD audio.