# Advanced Windows Capabilities with Multimedia

Author: Charles Calvert
Language: Borland Pascal for Windows
Location: 1993 Borland International Conference

# Advanced Windows Capabilities with Multimedia

Perhaps nothing that has happened in the computer world in the last few years has generated as much excitement as the new multimedia standards that have been developed by Microsoft, Apple and other major players in the industry.

Only ten years ago, considerable excitement was engendered by PC based programs that used simple text mode graphics characters to draw charts, graphs, and bouncing balls on the screen. But now programmers want the latest technologies which flood the senses with music, motion pictures, and thousands of colors.

Of course, all of these wonders are not going to come free of charge. They require investments in new equipment, new software, and above all, new programming skills.

Despite initial pessimism by industry wags, users have proved ready to pay the price necessary to bring multimedia onto the desktop. Microsoft, for instance, reports that one million new copies of Windows are installed on computers each month. Certainly, it wouldn't come as a major surprise to find out that this figure is somewhat exaggerated, but the trend would still be significant even if the numbers were only one tenth as high. On the hardware front, prices have plummeted to the point where users can pick up a fully decked out 486 screamer for under $2000.

Presumably, programmers are ready too, though of course we cast a somewhat warier eye on the phenomenon, since we are the ones who must somehow come to terms with the considerable technical intricacies inherent in the subject. It is, in fact, specifically those technical intricacies which are the subject of this paper. In particular, it focuses on the world of multimedia sound as represented by wave files, midi files, and CD players.

## The Tools and Definitions

Papers of this type usually dive right into the code itself, with little delay. But multimedia tools are so new, and the technology involved so daunting, that it would perhaps be helpful to spend a few moments clarifying certain issues.

The first to come to grips with is MPC, or Multimedia PC. This is a hardware

standard meant to define the minimum configuration of any personal computer that can play multimedia programs. The key elements in the MPC standard are a 386 or better computer with a VGA video system, a mouse, a sound board, and a CD-ROM.

The programs in this paper were tested on two 486s, one equipped with Creative Labs' Sound Blaster Pro card and the other with a NEC CD-ROM drive. The first is a requirement for playing MIDI files, and the second for playing CDs.

If you don't have either of these tools then you will not be able to run any of the code accompanying this paper. It is important to understand that the widely distributed Speaker.drv file will not be enough to allow you to run the included code. The reason for this is that the calls used in this paper are too low level for the limited WAV file support available via Speaker.drv.

Now that we've clarified the hardware requirements, we can move on to a discussion of the various devices referenced in this paper. It is probably safe to assume that everyone now knows what a CD-ROM is, but there may be some lingering confusion about WAV and MIDI files.

WAV files are a Microsoft standard file format generally used for recording non-musical sounds such as the human voice or a car horn. The key fact to know about them is that they can store about one second of sound in 11K of disk space, or one minute of sound in one meg of disk space. As a result, this medium tends to be extremely disk intensive, and is used mostly for adding short sound effects to a program or to the entire Windows environment.

The word MIDI stands for Musical Instrument Digital Interface. Put in the simplest possible terms, MIDI files can be thought of as containing a series of notes such as C sharp or a A flat which are sent to a synthesizer with instructions to play that note using the sounds associated with a particular instrument such as a piano, horn or guitar.

The synthesizer I used when writing my paper came as a standard part of the Sound Blaster Pro card. Most sound cards and MIDI files can play between 6 and 16 notes at once, and can imitate between 3 and 9 instruments. MIDI files can store one minute of fairly high quality musical sound in about 5K of disk space. This means that they are much more useful than WAV files for most computer users.

In this paper, I will not discuss how to record or mix MIDI or WAV files. Instead, I will concentrate on showing how MIDI and WAV files can be played using the Media Control Interface (MCI).

# Narrowing the Focus

Now that we know something about the media being discussed in this paper, it is time to turn our attention to the programming techniques used to make a computer generate sounds.

As it happens, Microsoft provides three separate interfaces that can allow you to access multimedia devices. Two of these are part of MCI, while the third is a low-level API which tends to be very rigorous and demanding.

Microsoft has stated publicly that it will not promise to support the low-level API in the future. Since few programmers can afford to spend time learning a standard which is fleeting at best, we can safely ignore this rather challenging subject. (Phew!)

With the low level API out of the way, that leaves only two remaining programming techniques. The first is a string based interface meant primarily to provide support for very high level languages such as Visual Basic. Since we are working in Pascal and C, we have access to the third technique, which is a powerful message based interface.

The peculiar thing about this command-message interface is that it relies very heavily on a single routine called MciSendCommand, which takes four parameters. Though this might sound fairly limiting at first, in practice it turns out to be a fairly flexible system.

The first parameter passed to MciSendCommand is a handle or id number used to identify the particular device in question. For instance, when you first open up a CD drive, you pass 0 in this parameter, since you don't yet have an id for the device. But thereafter you pass in the id which was returned to you when you opened the device.

The key to this entire interface is the second parameter passed to the function, which is a message which conveys a particular command. Here is a list of the twelve most common of these messages, along with its meaning:

| | |
|---|---|
| **Mci_Capability** | **Query the devices abilities** |
| **Mci_Close** | **Close a device** |
| **Mci_Info** | **Query type hardware being used** |
| **Mci_Open** | **Open a device** |
| **Mci_Play** | **Play a song or piece on a device** |
| **Mci_Record** | **Record to a device** |
| **Mci_Resume** | **Resume playing or recording** |
| **Mci_Seek** | **Move media forward or backward** |
| **Mci_Set** | **Change the settings on device** |
| **Mci_Status** | **Is device paused, playing, etc.** |
| **Mci_Stop** | **Stop playing or recording** |

To complement these commands there is a set of flags and records available which can give programmers the kind of fine tuning they need to get the job done right. For instance, the Mci_Play message has four important flags that can be OR'd together to form its third parameter:

| | |
|---|---|
| **Mci_Notify** | **Post Mm_Notify message on completion** |
| **Mciy_Wait** | **Complete operation before returning** |
| **Mci_From** | **Starting position is specified** |
| **Mci_To** | **Finish position is specified** |

The last two flags presented above can be OR'd together like this:

**Mci_From or Mci_To**

In this form, they inform MCI that a starting and finishing position will be specified in the last parameter.

The fourth parameter is a pointer to a record, or more commonly, the address of

the record itself. Most of the time, the record passed via an MciSendCommand will differ depending on the message being sent. In the above case, when an Mci_Play message is being sent, the structure looks like this:

```
PMci_Play_Parms = ^TMci_Play_Parms;
TMci_Play_Parms = Record
  dwCallback: LongInt;
  dwFrom: LongInt;
  dwTo: LongInt;
end;
```

Sometimes it is necessary to fill out all three fields of this structure, and at other times, some, or none of them can be filled out. For instance, if you set the Mci_Notify flag in the second parameter of MciSendCommand, then you probably will want to set dwCallback equal to the HWnd of the Window you want MCI to notify. Specifically, if you were inside a TDialog descendant at the time you started playing a WAV file, then you would want to pass the dialog's HWindow in dwCallback, so that the dialog would be informed via an Mm_Notify message when the WAV file stopped playing.

Of course, if you set both the Mci_From and Mci_To flags, then you would want to fill out both the dwFrom and the dwTo fields of the TMci_Play_Parms record. And so on. Remember that the TMci_Play_Parms recorded is associated explicitly with the Mci_Play message. Other messages have their own unique record structures. For instance, the Mci_Open message is associated with the TMci_Open_Parms structure.

All of the multimedia structures or constants discussed in this paper are listed in both the MmSystem.hlp file and MmSystem.pas. You should definitely take the time needed to become familiar with these files. Note also, that in the BPW 10/23/92 release, the MmSystem.hlp file allows you to search for a structure only via the "C" language syntax, that is, without prefixing the name with a "T". Therefore you should search for MCI_OPEN_PARMS, not TMci_Open_Parms.

The only major aspect of the MciSendCommand function that we haven't discussed yet is its return value, which happens to be an error number kept in the low order word of a LongInt. Microsoft comes through with a nice touch at this point, by adding the MciGetErrorString function, which provides you with an explanation of any error in return for the result of MciSendCommand function. MciGetErrorString will even send you back a pleasant little message that all has gone well, if that is indeed the case.

# Coming to Terms with MciSendCommand

If you have never dealt with an interface like the one described above, then it is possible that a few words of explanation are in order. The goal here is to totally encapsulate the multimedia API inside a message-based system that isolates multimedia programmers from the details of the code base's actual implementation. In other words, when using this mid-level interface, there is no point at which you or I would actually call a true multimedia API function.

The reason for this is one that should be familiar to all object-oriented

programmers. Specifically, hardware and operating systems change over time, and as a result APIs are forced to change with them. When API's change, then existing code bases are rendered obsolete, and last year's work has to be done all over again.

But the MCI command-message interface protects the programmer from any fluctuations in the API. For all practical purposes, all we are doing is sending messages into a dark hole. What goes on inside of that hole is of little concern to us. Five year's from now CD's may have doubled their capacity and cut their access time down to a fifth of its current snail-like pace. But none of that is going to affect our code. All we do is say that we want a particular track to be played. How its played is of no concern to us.

Another crucial advantage of this style of programming is that it gives the user a common interface to a series of radically different pieces of hardware. For instance, at this time the MCI command interface works with the following different types of devices, which are listed here opposite their official MCI name:

| | |
|---|---|
| **animation** | **Animation device** |
| **cdaudio** | **CD player** |
| **dat** | **Digital audio tape device** |
| **digitalvideo** | **Digital video device** |
| **scanner** | **Image scanner** |
| **sequencer** | **MIDI** |
| **vcr** | **Video tape player** |
| **videodisc** | **Videodisc device** |
| **waveaudio** | **A device that plays WAV files** |

What MCI has tried to do is to find the things that all these devices have in common, and then to use these similarities to bind them together. In particular, it makes sense to ask all of these devices to play something, to stop playing, to pause, to seek to a particular location in their media, etc. In other words, they all respond to the set of commands listed above as the primary MCI messages.

Talk about device independence! The Windows multimedia extensions not only protect you from the details of how a particular device might work, but they also frequently allow you to treat one type of device exactly the same way you would treat another, very different, type of device. For instance, you can come very close to using the exact same code to play a WAV file as you would to play a MIDI. The only difference would be that one time you would tell MCI that you want to work with a "waveaudio" device, while the next time you would say that you want to work with a "sequencer".

# A Note on Style

Before moving on, I should perhaps mention something about the conventions I've adapted when writing MCI code. Here for instance, is one traditional way to tell MCI to play the middle third of a Wave file that is exactly 600 milliseconds in length:

```
procedure PlayWave;
var
  Mci_Play_Parms: TMci_Play_Parms;
  dwReturn: LongInt;

begin
  Mci_Play_Parms.dwCallback := HWindow;
  Mci_Play_Parms.dwFrom := 200;
  Mci_Play_Parms.dwTo := 400;

  dwReturn := MciSendCommand(wDeviceId, Mci_Play,
              Mci_Notify or Mci_From or Mci_To,
                                  LongInt(@Mci_Play_Parms);

  if dwReturn <> 0 then HandleError(dwReturn);
end;
```

Procedures that look very similar to this will be repeated over and over again throughout an application, as each command is issued.

I attempt to clarify this code somewhat by always using *Info* as the identifier for the record involved, and by always passing a variable called *Flags* as the second parameter. Once these changes are made, the preceding code looks like this:

```
procedure PlayWave;
var
  Info: TMci_Play_Parms;
  Flags,
  R: LongInt;

begin
  Info.dwCallback := HWindow;
  Info.dwFrom := 200;
  Info.dwTo := 400;

  Flags := Mci_Notify or Mci_From or Mci_To,
  R := MciSendCommand(wDeviceId, Mci_Play, Flags, LongInt(@Info);

  if R <> 0 then HandleError(dwReturn);
end;
```

Though still somewhat less than a paradigm of clarity, I find material presented in this way to be fairly easy to manage.

# The Player Program

To help illustrate the points made in this paper I have constructed an example program called Player that will work with CDs, WAV files, and MIDI files.

The "Player" program is divided into two major sections: a main file and a set of DLLs. Specifically, there are three DLLs, one containing the MCI code for playing WAV files, one containing the code for playing MIDI files, and the other for playing CDs.

All three of these DLLs use a common unit which encapsulates most of the functionality involved with the actual playing of the various files. The reason why all three DLLs are able to share so much code is explained above, when I described the extreme device independence of the MCI interface.

Specifically, here are the function headers from the interface to the PlayInfo unit, which is shared by all three DLLs:

```
function CloseMCI: Boolean; export;
function ErrorMsg(Error: LongInt; Msg: PChar): Boolean; export;
function GetDeviceID: Word; export;
function GetInfo(S: PChar): PChar; export;
function GetLen: Longint; export;
function GetLocation: LongInt; export;
function GetMode: Longint; export;
function OpenMCI(PWindow: HWnd; FileName, DeviceType: PChar): Boolean;
export;
function PlayMCI: Boolean; export;
function SetTimeFormatMs: Boolean; export;
function StopMci: Boolean; export;
```

Most of the time, all three devices can use the same code to play a file, close a file, stop a file, get the length (in milliseconds) of a file, and to pause a file.

The open command, however, needs to be customized for each device. Its first parameter is the HWindow of the dialog, window or control which is playing the file. The second parameter is the name of the file and the third is the device type. Most of the major device types, such as sequencer, waveaudio and videodisc, are listed above. Obviously, the last parameter is in some ways the most important, as it is the one which tells MCI whether we want to play a CD, a WAV file, or a MIDI file.

The other half of the program, the part that is not a DLL, is written in standard OWL code. It consists of a main Window which tells the user about the system's multimedia capabilities, and also a user interface for the code in the DLLs.

Player finds out about the capabilities of the current system by iterating through the following array of possible types of devices to create a series of buttons and checkboxes with the appropriate name as window text:

```
MMTypes:array[0..10] of PChar = ('cdaudio', 'dat',
                                 'digitalvideo', 'MMMovie', 'other',
                                 'overlay', 'scanner', 'sequencer',
                                 'vcr', 'videodisc', 'waveaudio');
```

Next to these controls is a second set of checkboxes with window text made from the following array:

```
MMAbles:array[0..10] of PChar =
          ('Not Supported', 'Can Eject', 'Can Play',
           'Can Pause', 'Can Stop', 'Can_Record',
           'Can Save', 'Is Compound', 'Has Audio',
           'Has Video', 'Uses Files');
```

When Player starts, the checkboxes associated with a particular device are marked if the current machine supports that device. If the user selects the button with that device's name on it, then Player tells the user if the device can eject, play, pause, stop etc. It queries Windows to obtain this information by passing in one of the following flags in connection with the Mci_GetDevCaps function:

```
Tests:array[1..10] of LongInt =
          (Mci_GetDevCaps_Can_Eject,
           Mci_GetDevCaps_Can_Play,
           Mci_GetDevCaps_Can_Play,
           Mci_GetDevCaps_Can_Play,
           Mci_GetDevCaps_Can_Record,
           Mci_GetDevCaps_Can_Save,
           Mci_GetDevCaps_Compound_Device,
           Mci_GetDevCaps_Has_Audio,
           Mci_GetDevCaps_Has_Video,
           Mci_GetDevCaps_Uses_Files);
```

Here is the way the code in question actually appears in the program:

```
Info.dwItem := Tests[i];
Flags := Mci_GetDevCaps_Item or Mci_Notify;
Result := MciSendCommand(id, Mci_GetDevCaps, Flags,
                  LongInt(@Info));
if Info.dwReturn > 0 then
  SendMessage(BoxAble[i]^.HWindow, BM_SetCheck, 1, 0);
```

As you can see, the program sends a BM_SetCheck message to the appropriate checkbox if the ability is supported on the users system. All of the above lines appear inside a loop which allows the code to check for each capability in turn.

The end result of the above activities is to inform the user immediately what devices are available on his or her system, and then what type of abilities are associated with a particular device. As a final service, the program queries the users System.ini file to obtain the names of the currently selected multimedia drivers.

Along the top of the main window is a menu which allows the user to open one of three dialogs, the first for running a CD player, the second for MIDI files, and the third

for WAV files.

Each of these dialogs comes with buttons that allow the user to start, stop and pause the relevant device. The rest of the dialogs display information about the device in question, and about the current length and format of any file the user might choose to play.

Internally, the Player program consists of a module called Player.pas which controls the main window, and three sub-modules, called WavePlay, MidiPlay and CDPlay, each of which controls a dialog. Any routines that can be shared in common between all three sub-modules are placed inside a file called PlayDlg.

It goes without saying that the material in the DLLs would lend itself nicely to an object oriented format, but I decided to forgo that luxury in order to reap the considerable benefits derived from using DLLs.

In general, you should be aware of the steps that need to be taken whenever you open up a multimedia file. The first two steps are to find out if any existing hardware is available and to open the device itself. If either or both of these steps fail, then you need to exit as gracefully as possible. Because of the mciGetErrorMessage function, it is easy for you to post an appropriate error message for the user.

After opening up the file, the program reports on its length and format, and then begins to play it. While the user is listening to the file, Player reports on the file's progress, which is particularly important when playing CD or MIDI files, which can last for several minutes or more.

When the file has stopped playing, or if the user has aborted the play, then the program closes the device before exiting. At all times, you should be checking the results of your calls, so that you can be aware if an error occurs.

Overall, these steps are not as demanding as they might seem when you first hear them. Certainly in some circumstances it might be appropriate for you to skip some of them, but you should be aware of the general scheme, and the way it fits together.

# Details

At this point, all that remains to be covered are a few details which might cause confusion to the reader. In particular, you should notice the function called SetTimeFormatMS:

```
function SetTimeFormatMS: Boolean;
var
  Info: TMci_Set_Parms;
  Flags,
  Result: LongInt;
  S1: array [0..MsgLen] of Char;
begin
  SetTimeFormatMS := True;
  Info.dwTimeFormat := Mci_Format_Milliseconds;
  Flags := Mci_Set_Time_Format;
  Result := MciSendCommand(wDeviceID, MCI_Set,
```

```
                    Flags, LongInt(@Info));
  if Result <> 0 then begin
    ErrorMsg(Result, S1);
    SetTimeFormatMS := False;
  end;
end;
```

This code is very much like the PlayWave procedure shown above, but instead of using a TMci_Play_Parms record, it uses a TMci_Set_Parms record:

```
  TMCI_Set_Parms = record
    dwCallback: Longint;
    dwTimeFormat: Longint;
    dwAudio: Longint;
  end;
```

The key member of this structure is dwTimeFormat, which is used to select a particular time format. The Player program uses milliseconds, though I could have chosen seconds and minutes, or even the number of tracks that have been played.

Notice that SetTimeFormatMS receives Mci_Set_Time_Format as the third parameter. Other messages I could have passed in its stead include Mci_Set_Door_Closed or Mci_Set_Door_Open. This latter flag can be used to eject a CD from a CD player.

Some folks might not find it intuitively obvious to search out the Mci_Set message as the place to issue the command to eject a cassette. This highlights one possible criticism of the MCI command interface, namely that it lacks some of the intuitive feel of an API which might feature a command such as "EjectCD".

One final point involves the posting of Mm_Notify messages to the dialog objects in the main program. These messages are routed to standard Pascal message response functions, such as this one from PlayMIDI.pas:

```
procedure TMidiDlg.MciNotify(var Msg: TMessage);
begin
  KillTimer(HWindow, MidiTimer);
  ReportStatus;
  if Mode = Mci_Mode_Stop then CloseMci;
end;
```

Obviously the above code implies the presence of a Timer. The Timer is used to check on the status of the device being played. For instance, if a MIDI file is being played, then the timer allows us to check up on its progress at set intervals. In this particular program the intervals are one quarter second in duration.

To fully understand the Mci_Notify method, you have to understand that Player can ask to receive a message whenever anything important happens to the file being played. For instance, if the file ends, or if the user presses the Pause button, then an Mm_Notify message is posted. Here is the declaration for the MciNotify function listed

above:

```
procedure MciNotify(var Msg: TMessage);
   virtual wm_First + mm_MciNotify;
```

If the file currently being played is finished, or if the user has asked to abort the play, then the proper response is to close the device. But of course this is not what we want to do if the user has simply paused the file. To distinguish between these two different events we process a Mci_Status message in the main program with the following code :

```
procedure TMidiDlg.ReportStatus;
var
  S: array[0..MinLen] of Char;
begin
  Mode := GetMode;
  case Mode of
    Mci_Mode_Not_Ready: StrCopy(S, 'Ready');
    Mci_Mode_Pause: StrCopy(S, 'Pause');
    Mci_Mode_Play: StrCopy(S, 'Play');
    Mci_Mode_Stop: StrCopy(S, 'Stop');
    Mci_Mode_Open: StrCopy(S, 'Open');
    Mci_Mode_Record: StrCopy(S, 'Recording');
    Mci_Mode_Seek: StrCopy(S, 'Seeking');
  end;
  SStatus^.SetText(S);
end;
```

This procedure first gets the current mode of the device by calling the following procedure, which is located in PlayInfo.pas file, a support file used by all of the DLLs which accompany Player.pas:

```
function GetMode: Longint;
var
  Info: TMci_Status_Parms;
  Flags,
  Result: LongInt;
  S1: array [0..MsgLen] of Char;
begin
  FillChar(Info, SizeOf(TMci_Status_Parms), 0);
  Info.dwItem := Mci_Status_Mode;
  Flags := Mci_Status_Item;
  Result := MciSendCommand(wDeviceID, Mci_Status,
                    Flags, LongInt(@Info));
  if Result <> 0 then begin
    ErrorMsg(Result, S1);
```

```
    exit;
  end;
  GetMode := Info.dwReturn;
end;
```

In my opinion, the GetMode function highlights the opacity to which the MCI command interface can sometimes fall prey. Unless you actually looked up the Mci_Status, Mci_Status_Item and Mci_Status_Mode messages in the MCI reference books, it is unlikely that you would guess exactly what this function does just from looking at it.

Nevertheless, it does very effectively meet our current needs. After calling it, we know whether the device is paused or stopped, and so we can handle the Mm_Notify message with the appropriate response. In particular, I first set a variable called Mode to the value returned from GetMode, and then show the user a string which explains the result of the query. The string is displayed in a static text Control which has been inserted in the object's dialog.

# Summary

Though it has not been possible for me to explore all of the many aspects of the MCI interface in this paper, still I believe I have been able to give you enough information to get you up and running. If you want to learn more, the best thing you can do now is study the included example and MmSystem.pas interface unit which shipped with BP. As soon as you have some feeling for how this code works, then you should begin writing your own code for manipulating multimedia files.

Overall, I think you will find the MCI interface a flexible and well structured tool, that appears to be constructed so that it will withstand the ravages of time with a fair degree of aplomb. Certainly its weaknesses are more than made up for by the excitement of multimedia programming. So go to it with a will and be sure to take the time to enjoy yourself.