Tips & Tricks

Memory Mapped Files

A lot of work by programmers is concerned with the handling of files. Sometimes you only read files, sometimes you write back to them. You will either use the `file` or `text` types, or perhaps a simple stream like `TFileStream`. You will always need to position the file pointer before you read or write.

In Win32 there is an API for *memory mapped files*. Now you can open a file and access its contents just as if they were all in memory. Technically, this is done with the Virtual Memory Manager (VMM). Access via a normal memory pointer has several advantages against normal file access routines.

As an example let me demonstrate the use of a memory mapped file for a patch program. The patch program needs to find a specifc position in a file, read the contents, fix it and write it back to the file. The file can be large: for example a Delphi .EXE file. First, let me introduce a method for searching for a string in a huge buffer: the Boyer Moore method. It searches very fast if it can access memory directly. See the code in Listing 1. Now let us assume that you have defined the following constant in the application whose .EXE file you want to patch:

```
type
  PPatch = ^TPatch;
  TPatch = packed record
    Id : String[9];
    Serial : Longint;
  end;
const
  MyPatch : TPatch =
    ( Id : '*MyPatch*'; Serial : 0);
```

So, we want to look for the string `*MyPatch*` in a file and patch the serial number code to, say, 4711. To have easy access to the memory-mapped file I use a descendant of `TFileStream` with the new `Memory:Pointer` property. See the code in Listing 2. As you can see it creates a read only or read/write file mapping depending on the `Mode` parameter. The destructor releases the previous file mapping.

Our patching code is now very simple, using the new stream type together with the Boyer Moore search. The code opens the memory mapped file stream and then calls the string search on the memory pointer. If the search is successful it does some pointer arithmetic to obtain the position in the file. The -1 is neccessary because the search does not work with the length byte

➤ *Listing 1*

```
{ Boyer Moore string search on a buffer }
function PosBM(const P: String; const Buffer;
  Count: Longint): Longint;
var
  i,j   : Cardinal;
  m,n   : Cardinal;
  Skip : array[Char] of Integer;
  procedure InitSkip;
  var
    Ch: Char;
    i: Integer;
  begin
    for Ch := Low(Char) to High(Char) do Skip[Ch] := M;
    for i := 1 to M do Skip[P[i]] := M-i;
  end;
  function A(Index: Cardinal): Char;
  begin
    Result := Chr(TByteArray(Buffer)[Index-1]);
  end;
begin
  Result := 0;
  if (Count=0) or (P='') then exit;
  M := Length(P);
  N := Count;
  i := M; j := M;
  InitSkip;
  repeat
    if (A(i)=P[j]) then begin
      Dec(i);
      Dec(j);
    end else begin
      if M-j+1 > Skip[A(i)] then
        i := i+M-j+1
      else
        i := i+Skip[A(i)];
      j := M;
    end;
  until (j<1) or (i>N);
  if i>N then
     Result  :=  -1
  else
     Result := i;
end;
```

➤ *Listing 2*

```
{ Stream for memory-mapped files }
type
  TMemoryFileStream = class(TFileStream)
  private
    FFileMapping : THandle;
    FFileBase    : Pointer;
  public
    constructor Create(const FileName: String;
      Mode: Word);
    destructor Destroy; override;
    property Memory: Pointer read FFileBase;
  end;
constructor TMemoryFileStream.Create(
  const FileName: String; Mode: Word);
var
  aFlag : Integer;
begin
  inherited Create(FileName, Mode);
  if (Mode and fmOpenReadWrite) <> 0 then
    aFlag := PAGE_READWRITE
  else
    aFlag := PAGE_READONLY;
  FFileMapping :=
    CreateFileMapping(Handle, nil, aFlag, 0, 0, nil);
  if FFileMapping = 0 then
     raise Exception.Create(
       'CreateFileMapping failed');
  if (Mode and fmOpenReadWrite) <> 0 then
    aFlag := FILE_MAP_WRITE
  else
    aFlag := FILE_MAP_READ;
  FFileBase :=
    MapViewOfFile(FFileMapping, aFlag, 0, 0, 0);
  if FFileBase = nil then
    raise Exception.Create('MapViewOfFile failed');
end;
destructor TMemoryFileStream.Destroy;
begin
  if FFileBase <> nil then
    UnmapViewOfFile(FFileBase);
  if FFileMapping <> 0 then
    CloseHandle(FFileMapping);
  inherited Destroy;
end;
```

of the string but the length byte is included in the TPatch record! Writing the new data into the file is simple. See Listing 3.

Think about what you have to do without using a memory mapped file: you have either to copy the file data into a memory block or you have to re-write the search so it works especially on a file or stream. Both methods make for more coding work and I think the performance of memory mapped files is great.

Contributed by Stefan Boether (author of XTools) of Fabula Software, Germany, email stefc@fabula.com

Smart Drag And Drop

I find the drag and drop options in Windows 95 really nice – it's easier than opening a file from dialog. Unfortunately, not all applications support it. To implement drag and drop in your application you must do the following:

- Allow a window handle to accept drag messages,
- Respond to the WM_DROPFILES message,
- Release the received drop handle.

In the form's OnCreate handler you must call the API function DragAcceptFiles(Handle, True) with the window handle that needs to accept the dragged files. Than you can catch the message WM_DROPFILES and after this free the handle with a call to DragFinish(FDrop).

But here's some extra tricks to make it a lot easier. Everything is done in a component, which also converts the Windows WM_DROPFILES message to a Delphi compliant DragDrop event. So you can use the normal OnDragDrop event which is defined by each control. The component is called TxFileDrop.

The first problem is how to catch the WM_DROPFILES the *form*, and not my non-visual component, receives. I solve this with a hook to the Windows procedure of the component owner that must be the TForm in which I place it. This is done with the call:

```
FOldWndProc := TFarProc(SetWindowLong(
  Handle, GWL_WNDPROC,
  Longint(MakeObjectInstance(FormWndProc))));
```

Now I can get all the windows messages for the form first, but all the messages I not interested in I must send to the previous message handler. The two methods shown in Listing 4 do this. DefaultProc calls the old handler we saved in FOldWndProc. FormWndProc handles the WM_DROPFILES messages and calls wmDropFiles for it.

But such tricky hooks must be removed otherwise the system can be crashed. To do this I install a second daisy chain mechanism on the OnDestroy event of the owner form:

```
FNextDestroy := OnDestroy;
OnDestroy := FormDestroy;
```

In this case before the form is destroyed the following method is called. I can't handle it in the destructor because the window handle of the owner form must be valid. After switching the windows handler back to the prior call the next destroy event is called if assigned:

```
procedure TxDropFile.FormDestroy(Sender:TObject);
begin
  with TForm(Owner) do
    FreeObjectInstance(Pointer(SetWindowLong(
    Handle, GWL_WNDPROC, Longint(FOldWndProc))));
  if Assigned(FNextDestroy) then
    FNextDestroy(Sender);
end;
```

Now the framework for our component is ready. We can move on to handle the drop event and the calling of wmDropFiles. First we need to access the handle for the drop. It's stored in the wParam of the message. Also we need the point on which the drag occurs. This client form related point we convert to screen coordinates for later use:

```
FDrop := Msg.wParam;
DragQueryPoint(FDrop,aPoint);
aPoint := TForm(Owner).ClientToScreen(aPoint);
```

After this we look to see how many files are dropped and step through the list, inserting the full pathname of each file into our file list (which is a stringlist to make Delphi access easy):

```
FFiles.Clear;
aFiles := DragQueryFile(FDrop, $FFFFFFFF, Nil, 0);
for i := 0 to aFiles-1 do begin
  aLen := DragQueryFile(Msg.wParam, i,nil,0);
  DragQueryFile(Msg.wParam, i, aFilename, aLen+1);
  FFiles.Add(StrPas(aFileName));
end;
```

➤ *Listing 3*

```
procedure PatchExe(const aFile: String; aSerial: Longint);
var
  aPos    : Longint;
  aStream : TMemoryFileStream;
begin
  aStream :=
    TMemoryFileStream.Create(aFile, fmOpenReadWrite);
  try
    aPos :=
      PosBM('*MyPatch*',aStream.Memory^, aStream.Size);
    if aPos <> - 1 then
      { - 1 for Length byte ! }
      PPatch(PChar(aStream.Memory)+aPos-1)^.Serial :=
        aSerial;
  finally
    aStream.Free;
  end;
end;
```

➤ *Listing 4*

```
procedure TxDropFile.DefaultProc(
  var Message:TMessage);
begin
  with Message do
    Result := CallWindowProc(FOldWndProc,
      TForm(Owner).Handle, Msg, wParam, lParam);
end;
procedure TxDropFile.FormWndProc(var Message:TMessage);
begin
  if Message.Msg = WM_DROPFILES then wmDropFiles(Message)
  else DefaultProc(Message);
end;
```

Now the best trick of the component cames: we convert the Windows event to a real Delphi DragDrop operation. First we must find the control that is at the given coordinate. Delphi gives us the method FindDragTarget which acts on screen coordinates – this is the reason why we previously converted the client coordinates to screen coordinates. If we find such a control we convert the point again to the coordinate space of the control and call the DragDrop method of the control with our component as the sender and the given point:

```
aControl := FindDragTarget(aPoint,False);
if Assigned(aControl) then begin
  aPoint := aControl.ScreenToClient(aPoint);
  aControl.DragDrop(Self,aPoint.X,aPoint.Y);
end;
```

All you now must do to receive such a drop operation is to react to the OnDragDrop event and look to see

```
if Source is TxFileDrop
```

For example, we can drop a file list into a memo with the following code:

```
procedure TForm1.Memo1DragDrop(
  Sender, Source: TObject; X, Y: Integer);
begin
  if Source is TxDropFile then
     Memo1.Lines := xDropFile1.Files;
end;
```

Isn't this easy? Just place the new component on your form and then use normal Delphi events.

Contributed by Stefan Boether (author of XTools) of Fabula Software, Germany, email stefc@fabula.com

Listbox Keyboard Miscellanea

It is usually a good idea to strive to make sure that any action which can be done by mouse can also be done with the keyboard. It's a given fact that drag and drop can't easily be mimicked, but the result of the drag and drop operation should be achievable through keystrokes alone.

Anyway, the point I'm getting around to is that non-contiquous listbox entry selection is performed by clicking on the first item and Ctrl+clicking on subsequent items. How do we do the same with the keyboard alone? Listboxes have always supported it, but for some reason the keystrokes weren't well advertised.

The answer is to press Shift+F8 (not particularly intuitive) which starts the highlight marker flashing. The arrow keys can then move it up and down and the Space bar toggles that item as selected and unselected. Another Shift+F8 gets out of this mode.

Of course, contiguous selection is not a problem in a listbox: Shift plus the arrow keys do that. Selecting the entire contents of a listbox can be done by Home, Shift+End, or End, Shift+Home, but a quicker (and lesser known) way is Ctrl+/, where Ctrl+\ un-selects all but the active item.

In a Windows 95 or Windows NT list view, most of these key strokes are changed. Ctrl+A selects all and discontiguous selections are easier. Hold the Ctrl key down, move around with the arrow keys and press Space to select an item.

Contributed by Brian Long

Creating A Wave File

Delphi's TMediaPlayer is a nice and powerful tool for handling many multimedia tasks. However, it does have a few quirks. One of them is that TMediaPlayer can only open a WAV file that has at least one byte of data in it. If you want to use TMediaPlayer to record a brand new, empty, wave file you cannot merely set the Filename property and start recording. TMediaPlayer expects Filename to address a valid WAV file that has some data in it already. Merely creating an empty file with a TWaveHeader in it won't do the trick either.

The code in Listing 5 creates a WAV file with a single byte of data at the beginning. It uses calls to the

➤ *Listing 5*

```
unit Makewave;
interface
function CreateNewWave(NewFileName: String): Boolean;
implementation
uses
  MMSystem, SysUtils, Windows;
function CreateNewWave(NewFileName: String): Boolean;
var
  DeviceID: Word;
  MciOpen: TMCI_OPEN_PARMS;
  MciRecord: TMCI_RECORD_PARMS;
  MciSave: TMCI_SAVEPARMS;
  MCIResult: LongInt;
  Flags: Word;
  TempFileName: array[0..MAX_PATH] of Char;
begin
  Result := True;
  { If anything fails along the way,
    it will turn False }
  StrPCopy(TempFileName, NewFileName);
  { Open the Device }
  MciOpen.lpstrDeviceType := 'waveaudio';
  MciOpen.lpstrElementName := '';
  Flags := MCI_OPEN_ELEMENT or MCI_OPEN_TYPE;
  MCIResult := MciSendCommand(0, MCI_OPEN, Flags,
    LongInt(@MciOpen));
  Result := Result and (MCIResult = 0);
  DeviceID := MciOpen.wDeviceId;
  { Record only one byte of data }
  MciRecord.dwTo := 1;
  Flags := MCI_TO or MCI_WAIT;
  MCIResult := MciSendCommand(DeviceID, MCI_RECORD,
    Flags, LongInt(@MciRecord));
  Result := Result and (MCIResult = 0);
  { Save the file }
  mciSave.lpfileName := TempFilename;
  Flags := MCI_SAVE_FILE or MCI_WAIT;
  MCIResult := MciSendCommand(DeviceID, MCI_SAVE,
    Flags, LongInt(@MciSave));
  Result := Result and (MCIResult = 0);
  { Return True if device can be successfully closed }
  MCIResult := MciSendCommand(DeviceID, MCI_CLOSE,
    0, LongInt(nil));
  Result := Result and (MCIResult = 0);
end;
end.
```

MMSYSTEM unit, specifically the MCISendCommand function. There are four steps involved. First, the wave audio device is opened. Then, the device is told to record a single bit of information. It will, technically, place one millisecond of input from your microphone into the wave, so watch what you say! Then the wave is saved with the given file name.

Finally, the device is closed. Along the way, any failure in the MCISendCommand calls will cause the Result of the function to be set to False.

The pattern for the four steps is relatively straightforward. First, the basic structures (the TMCI_*_Params structures) are filled out with the appropriate information. The Flags parameter is set and then the call to MCISendCommand is made, passing the Params structure as a LongInt created by casting the address of the structure.

There are a few extra things to note. The TMediaPlayer must be closed before the function is called, as this function makes direct calls to the wave audio device itself. You can re-open it after the WAV file has been created and its Filename property is set to the name of the new file.

The file name passed to CreateNewWave in the parameter NewFileName must contain complete path information or else the default directory will be used.

CreateNewWave does no error reporting. It will return False if any error occurs in the calls to MCISendCommand. There are a whole slew of error code values that can be returned by MCISendCommand. They can be found in the MMSYSTEM.HLP file in the DELPHI\BIN directory. In addition, it will also return False if you try to create a WAV file that already exists.

Contributed by Nick Hodges, CompuServe 72662,2307

Thanks for all your Tips, keep them coming in!

If you have any hints that you think will be of use to fellow Delphi developers, just drop them in an email to the Editor at 70630.717@compuserve.com