# The Delphi CLINIC

**Edited by Brian Long**

***Problems with your Delphi project?***
***Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com***

## Bitmap Puzzle

**Q** The sample images supplied with Delphi are quite handy as far as sample images go. However, when I look at the button images, they are not as I would expect. They have a disgusting background colour and seem to be made up from two pictures. How do they work?

**A** The layout you describe is designed for `TBitBtn` and `TSpeedButton` usage. The `Glyph` property of these objects can be given a bitmap with a few special characteristics. We will deal with background colour a bit later.

The bitmap given to the `Glyph` property can have 1, 2, 3 or 4 individual images within it. If there is more than one image, they must all be the same size in a horizontal row. Assuming four images are supplied, reading left to right, they are interpreted as shown in Table 1, as per the online help.

If only one bitmap is given, the components modify it accordingly to indicate the other states. For example, if you set a `TBitBtn`'s `Enabled` property to `True`, it will grey out the image using some standard algorithm. The fourth image is only useful in a speedbutton, as a `TBitBtn` does not remain in a depressed state.

If you supply a bitmap to one of these `Glyph` properties, the code in the component will examine the size. For example, if the bitmap is 8 pixels high and 24 pixels wide, it will decide that it is made of three images and so set the `NumGlyphs` property to 3. This is great, and very convenient if there are indeed 3 square images represented in the bitmap. However, if there are really supposed to be 2 images, each of which is 8 pixels high and 12 pixels wide, `NumGlyphs` will be set wrong and the image drawn on the button will be too small. You can explicitly set `NumGlyphs` to fix that.

The other aspect about the `Glyph` property is that it has a transparency option. In 16-bit Windows, there was no easy mechanism for generating bitmaps with transparent areas. So the `TBitmap` type added a minor kludge to enable transparency support. `TBitmap` has a `TransparentColor` property. Any pixel in the bitmap that has this colour will be considered a transparent pixel. The main background colour of the canvas that the bitmap is drawn on will be drawn instead of that pixel from the bitmap.

The value of `TransparentColor` comes from the colour of the very bottom left pixel in the bitmap. The tasteless olive background in the Inprise-supplied sample images is used to accentuate the transparent areas. Since the bottom left pixel of all these supplied bitmaps is the same as the bad background colour, all the bad colour will be transparent.

As a small example, the Inprise-supplied bitmap file DOOROPEN.BMP looks like Figure 1. Similarly, DOORSHUT.BMP looks like Figure 2. We have the left image in each case being used when the mouse is up or down, and the second image being used when the relevant button is disabled. But the image could be more dynamic by combining the two. If you use MS Paint, or Borland Image

➤ *Table 1*

| Image Position | Button State | Description |
|---|---|---|
| First | Up | This image appears when the button is up (unselected). This image is also used when the button has focus (for example, if the user tabs to it); in this case, a focus rectangle is drawn around the button. If no other images exist in the bitmap, bit buttons also use this image for all other states. |
| Second | Disabled | This image usually appears dimmed to indicate that the button can't be selected. |
| Third | Clicked | This image appears when the button is clicked. The Up image reappears when the user releases the mouse button. |
| Fourth | Down | This image appears when the button stays down (indicating that it remains selected). |

➤ *Figure 1*

➤ *Figure 2*

➤ *Figure 3*

```
unit TblEdit1;
interface
procedure Register;
implementation
uses
  DBTables, DsgnIntf, Dialogs, SysUtils;
type
  TTableEditor = class(TComponentEditor)
    function GetVerbCount: Integer; override;
    function GetVerb(Index: Integer): string; override;
    procedure ExecuteVerb(Index: Integer); override;
    procedure Edit; override;
  end;
function TTableEditor.GetVerbCount: Integer;
begin
  Result := 3;
  ShowMessage(Format('Someone wants to know how many ' +
    'component editors we are adding. Well, the answer ' +
    'is %d', [Result]))
end;
function TTableEditor.GetVerb(Index: Integer): string;
begin
  { Someone wants the menu caption for one of
    our component editors }
  Result := Format('Component editor no. %d', [Index + 1])
end;
procedure TTableEditor.ExecuteVerb(Index: Integer);
begin
  ShowMessage('Someone has chosen one of our component ' +
    'editors. Here goes...');
  ShowMessage(GetVerb(Index))
end;
procedure TTableEditor.Edit;
begin
  ShowMessage(Format(
    'Someone double-clicked on a %s component called "%s"',
    [Component.ClassName, Component.Name]));
  { This would do the default behaviour, maybe manufacturing
    or locating an event handler:
      inherited Edit
    However, this will execute a component editor: }
  ExecuteVerb(0)
end;
procedure Register;
begin
  RegisterComponentEditor(TTable, TTableEditor)
end;
end.
```

➤ *Listing 1*

Editor, you can make an image that looks like Figure 3. Here the door is shut when the button is up (first image). When the user clicks the button, the door opens (third image). If the underlying button is disabled, the greyed out image is used. This is on this month's disk as DOORMOVE.BMP so you can check it out by using it for the `Glyph` property of a `TBitBtn` or `TSpeedButton`.

## TTable Component Editor

**Q** I am writing a new component inheriting from `TTable` and wish to add some extra component editors to it. Whenever I try to do this, I lose the original `TTable` component editors. How do I get around this?

**A** If you are unfamiliar with the terminology used in the question, then allow me to explain. When you right click on any component on a form designer, there are a standard set of menu items that appear in the popup menu, eg `Align To Grid`, `Send To Back`. However, some components have more entries on their popup menu.

A `TMainMenu`, for example, has an extra entry `Menu Designer...` and a `TTable` has various extra ones depending on the Delphi version. In Delphi 1, it just had `Fields Editor...`, but Delphi 2 and 3 have an additional `Explore` item (which invokes the Database Explorer). With a bare `TTable` (that is, with no `DatabaseName` or `TableName` properties) Delphi 4 looks just the same. However, if the `TTable` is set to point at an existing database table, Delphi 4 adds `Delete Table`, `Rename Table...` and `Update Table Definition`. If it points to a valid database, but to a non-existent table, you also get a `Create Table` option. Use the context-sensitive help available for these menu items to learn more about how they work.

All these menu items are called *verb*s, and are written in Delphi code in a *component editor* class. This is done in a manner not entirely dissimilar to writing property editors (discussed in *The Delphi Clinic* in the last issue). More information about component editors can be found by looking up the phrase on the *Collection '98 CD-ROM*, but, in short, these are designed to perform operations on a component more interesting than just modifying one property. In many cases they will modify several properties (such as the `TDatabase`'s component editor `Database Editor...`).

A simple component editor for a `TTable` is shown in Listing 1 (from TblEdit1.Pas on this month's disk). You install it just like installing a component. When installed, Delphi calls the `Register` routine and learns about the component editor's existence. You can see that a component editor object inherits from the Delphi-supplied `TComponentEditor` class, which is described in the help in all Delphi versions. `TComponentEditor` is defined in the DsgnIntf.Pas Open Tools API source file, supplied either in Delphi's SOURCE\VCL or SOURCE\TOOLSAPI directory, depending on the Delphi version.

When the user drops a component onto a form designer, Delphi identifies which component editor class is registered for it and creates an instance of that class. If no specific editor has been installed, Delphi uses a class called `TDefaultEditor`. When the user right clicks the component, it calls `GetVerbCount` to identify how many new verbs to add onto the context menu, then calls `GetVerb` sufficient times to get the menu item captions. If the user chooses one of these menu items, Delphi calls the component editor's `ExecuteVerb` method, passing in the relevant index.

If the user double clicks the component, the component editor's `Edit` method is called. If no specific component editor has been registered, `TDefaultEditor.Edit` attempts to identify the default event and makes an event handler for it (or jumps to that event handler if it already exists). Many component editors override `Edit` and make it execute one of the new verbs. Listing 1 executes the first verb, number 0.

The problem with this component editor, as mentioned in the question, is that all the normal `TTable` component editor verbs from the Delphi-supplied component editor are ignored if you install it. Delphi can only handle one component editor per

```
type
  TTableEditor = class(TComponentEditor)
  private
    FEditor: TComponentEditor;
  public
    constructor Create(AComponent: TComponent; ADesigner:
      {$ifdef DelphiLessThan4} TFormDesigner
      {$else} IFormDesigner {$endif}); override;
    destructor Destroy; override;
    function GetVerbCount: Integer; override;
    function GetVerb(Index: Integer): string; override;
    procedure ExecuteVerb(Index: Integer); override;
    procedure Edit; override;
  end;
const
  NewVerbs = 3;
  TableEditorClass: TComponentEditorClass = nil;
constructor TTableEditor.Create(AComponent: TComponent;
  ADesigner: {$ifdef DelphiLessThan4} TFormDesigner
    {$else} IFormDesigner {$endif});
begin
  inherited Create(AComponent, ADesigner);
  if TableEditorClass <> nil then
    FEditor :=
      TableEditorClass.Create(AComponent, ADesigner);
end;
destructor TTableEditor.Destroy;
begin
  FEditor.Free;
  FEditor := nil;
  inherited Destroy
end;
function TTableEditor.GetVerbCount: Integer;
begin
  Result := FEditor.GetVerbCount + NewVerbs;
  ShowMessage(Format(
    'Someone wants to know how many component editors ' +
    'we are adding. Well, the answer is %d', [Result]))
end;
function TTableEditor.GetVerb(Index: Integer): string;
begin
  { Someone wants the menu caption for one
    of our component editors }
  case Index of
    0..NewVerbs - 1 :
      Result :=
        Format('Component editor no. %d', [Index + 1])
  else
    Result := FEditor.GetVerb(Index - NewVerbs)
  end;
end;
procedure TTableEditor.ExecuteVerb(Index: Integer);
begin
  ShowMessage('Someone has chosen one of our component ' +
    'editors. Here goes...');
  case Index of
    0..NewVerbs-1 : ShowMessage(GetVerb(Index))
  else
    FEditor.ExecuteVerb(Index - NewVerbs)
  end;
end;
procedure Register;
var
  TmpTable: TTable;
  TmpTableEditor: TComponentEditor;
begin
  { Make a temporary table }
  TmpTable := TTable.Create(nil);
  try
    { What is the current component editor }
    TmpTableEditor := GetComponentEditor(TmpTable, nil);
    try
      { Make a note of its type }
      TableEditorClass :=
        TComponentEditorClass(TmpTableEditor.ClassType)
    finally
      TmpTableEditor.Free
    end
  finally
    TmpTable.Free
  end;
  { Now add our component editor to the foray }
  RegisterComponentEditor(TTable, TTableEditor);
end;
end.
```

➤ *Listing 2*

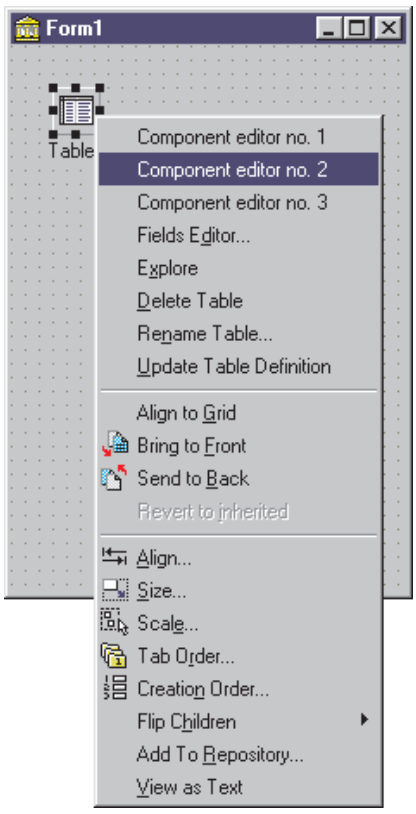
component, so how can we reconcile this problem? The answer I've found involves a bit of subterfuge.

Listing 2 (from TblEdit2.Pas) shows the important parts of a replacement `TTable` component editor. The underlying premise is to get hold of some reference to the original component editor class for `TTable` and store it. Then, the new component editor can call upon the old one whenever it needs to.

Unfortunately, we cannot just look through some source code, find the Delphi-supplied component editor class and refer to it. One reason is that, from version 3 of Delphi, Inprise have not supplied the relevant file (Delphi 1 and 2 users can find it as DBREG.PAS in Delphi's LIB directory). The more important reason is that, like Listing 1, DBREG.PAS defines the component editor class type, `TDataSetEditor`, in the `implementation` section of the unit and so we cannot gain access to it in code. This forces us into some underhand coding, so here goes.

In the `Register` routine, the code creates a `TTable` instance such that it can call a rather handy routine in the `DsgnIntf` unit. `GetComponentEditor` returns an instance of the original component editor. Rather than store this object directly, meaning we have a possibly unused object floating around for the whole Delphi session, we instead store a reference to its class type in an appropriate variable. Once the variable is set, the new component editor can be registered with the system. Of course, had we registered our component editor first, `GetComponentEditor` would have returned an instance of our class: not what we want.

When an instance of our component editor is created by Delphi, the constructor creates an instance of the original component editor for use if necessary. The destructor also ensures that it gets destroyed again. Now our new `GetVerbCount` can tell Delphi that we want our three verbs plus the number of verbs that the old component editor adds. Similarly, `ExecuteVerb` and `GetVerb` can extract information or invoke functionality from the old component editor as required. Figure 4 shows




➤ *Figure 4*

the modified context menu for a `TTable` in Delphi 4.

The only additional thing to mention here is the minor amount of conditional compilation used in

```
{$ifdef Ver90} { Delphi 2.0x }
  {$define DelphiLessThan4}
{$endif}
{$ifdef Ver93} { C++ Builder 1.0x }
  {$define DelphiLessThan4}
{$endif}
{$ifdef Ver100} { Delphi 3.0x }
  {$define DelphiLessThan4}
{$endif}
{$ifdef Ver110} { C++ Builder 3.0x }
  {$define DelphiLessThan4}
{$endif}
procedure TForm1.CheckBox1Click(Sender: TObject);
{$ifdef DelphiLessThan4}
const
  TVS_NOTOOLTIPS = $80;
var
  Style: Integer;
begin
  Style := GetWindowLong(TreeView1.Handle, GWL_STYLE);
  //Could use this if you just wanted to toggle states
  //Style := Style xor TVS_NOTOOLTIPS;
  if CheckBox1.Checked then
    Style := Style and not TVS_NOTOOLTIPS
  else
    Style := Style or TVS_NOTOOLTIPS;
  SetWindowLong(TreeView1.Handle, GWL_STYLE, Style);
end;
{$else}
begin
  TreeView1.Tooltips := CheckBox1.Checked
end;
{$endif}
```

➤ *Listing 3*

```
{$ifdef DelphiLessThan4}
function ExtractShortPathName(const FileName: string): string;
var
  Buffer: array[0..MAX_PATH] of Char;
begin
  SetString(Result, Buffer,
    GetShortPathName(PChar(FileName), Buffer, SizeOf(Buffer)));
end;
{$endif}
procedure TForm1.FileListChange(Sender: TObject);
begin
  if FileList.ItemIndex = -1 then Exit;
  lblLong.Caption := FileList.FileName;
  lblShort.Caption := ExtractShortPathName(lblLong.Caption);
  lblLongAgain.Caption := ExtractLongPathName(lblShort.Caption)
end;
```

➤ *Listing 4*

the component class definition. Delphi 4 changed the definition of one of the constructor parameters from an object reference of type `TFormDesigner` to an interface reference of type `IFormDesigner`.

## Treeviews And Tooltips

**Q** I have a big problem with treeviews. In a treeview, if the text of a node is too long, Windows automatically creates a hint window (a kind of tooltip) that overlaps the treeview control window when the mouse is moved around (like in Windows Explorer). Under Windows 95, everything goes fine. Under NT, once in a while the application terminates in an abrupt way and disappears from the task bar! Is there a way of solving this (for example, preventing Windows from creating the tooltip with the node text?)

**A** OK, so I'll précis that question into 'How can you turn automatic tooltips off in a treeview?' I have heard of a number of issues with the tooltip generation ability of treeviews in certain versions of the COMCTL32.DLL library, so I guess there might be a common need to do this.

It is worth emphasising that treeviews support two kinds of tooltips. Firstly, the standard VCL tooltip, controlled by the `Hint` and `ShowHint` properties. Whenever the mouse is paused anywhere over the control, if `ShowHint` is `True`, a tooltip will appear displaying whatever `Hint` was set to. If `Hint` is not set, the parent object's `Hint` is checked.

In addition, the treeview automatically provides a helpful tooltip whenever your mouse moves over a node with text that is partially obscured. This is much the same idea as that discussed in the article *Tooltips Under Your Control* in the last issue.

Delphi 4 makes the job of disabling these tooltips fairly easy: there is a `Tooltips` property that defaults to `True`. Under Delphi 3 or 2, you will need to do it manually. The Delphi 4 `Tooltips` property adds or removes the common control style `TVS_NOTOOLTIPS` from a treeview's window style as appropriate. `TVS_NOTOOLTIPS` is defined in Delphi 4 as a constant with a value of `$0080`, but did not exist in Delphi 2 and 3. Listing 3 shows some code that can be used to toggle treeview tooltips in an application (you can find the code in a simple application called TreeView.Dpr on the disk).

## Long And Short Filename Conversion

**Q** My query dates back to Issue 7 when you gave details of launching applications and waiting for them to terminate. I wrote a program using this which launched Excel Version 7 perfectly with a parameter of:

```
C:\MSOFFICE\OFFICE\LIBRARY\
  MYFILE.XLA
```

My user has now installed Excel 8, and the relevant parameter is now:

```
C:\PROGRAM FILES\
  MICROSOFT OFFICE\OFFICE\
  LIBRARY\MYFILE.XLA
```

This long file path is rejected by Excel which still seems to need the old short form:

```
C:\PROGRA~1\MICROS~2\OFFICE\
  LIBRARY\MYFILE.XLA
```

which works OK.

I thought I would write a 'simple' conversion routine to produce the short format. But this has turned out to be remarkably complicated. Note `MICROS~2` as the second slice above. My `Program Files` directory contains both `Microsoft Office` and `Microsoft Reference` subdirectories and it is possible that in addition to `PROGRA~1` there might be `PROGRA~2`.

Is there an easy way to convert the new long style path and filenames into their short equivalents?

**A** Extrapolating this question out a little, we have several

```
function ExtractLongFileName(const FileName: string):
  string;
var SearchRec: TSearchRec;
begin
  //Just in case something goes wrong
  Result := FileName;
  //Look for directories as well as just normal files
  if FindFirst(FileName, faDirectory, SearchRec) = 0
    then begin
    //If we find it, return the long name
    Result := SearchRec.Name;
    //If we don't have a long name, return the short name
    if Result = '' then
      Result :=
        String(SearchRec.FindData.cAlternateFileName);
  end;
  //Tidy up
  FindClose(SearchRec);
end;
```

➤ *Above: Listing 5*

➤ *Right: Listing 6*

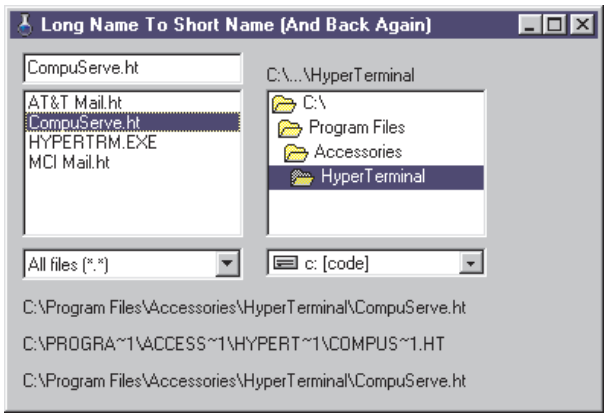
```
function ExtractLongPathName(const PathName: String):
  String;
var
  PathPart: String;
  Index: Word;
begin
  PathPart := PathName;
  { Get long name of the deepest level of path
    (file name or dir) }
  Result := ExtractLongFileName(PathPart);
  { Get long version of all directories in path }
  Index := Length(PathPart);
  repeat
    while (PathPart[Index] <> '\') and (Index > 4) do
      Dec(Index);
    if PathPart[Index] = '\' then begin
      PathPart[Index] := #0;
      Result :=
        ExtractLongFileName(PathPart) + '\' + Result
    end
  until Index = 4;
  { Finally add the drive letter }
  Result := Copy(PathName, 1, 3) + Result;
end;
```

questions to answer here. How does a 32-bit app translate between long and short file and path names, and also, how does a 16-bit app do the same (if it wants to)?

Let's tackle 32-bit applications first. The relevant code for this can be found in the LongToShort.Dpr project on the disk this month, and Figure 2 shows what it does. When you locate a file on the machine in the file listbox, the long filename is written on one label. This is then translated into the short name and written in a second label. Finally, the short version is translated back to a long version and written on a final label.

To accomplish the task, the program uses two translation routines, `ExtractShortPathName` and `ExtractLongPathName`. The former routine is a no-brainer in Delphi 4 as it is supplied in the `SysUtils` unit. Conditional compilation ensures that if the program is compiled under Delphi 2 or 3, a substitute routine that uses the `GetShortPathName` API is called instead. Listing 4 shows this routine and the code that updates the labels.

As you can see, going from long to short names is quite straightforward.




➤ *Figure 2*

➤ *Listing 7*

```
uses ShellAPI;
function ExtractLongFileName(const FileName: string): string;
var Info: TSHFileInfo;
begin
  if SHGetFileInfo(PChar(FileName), 0, Info, SizeOf(Info),
    SHGFI_DISPLAYNAME) <> 0 then
    Result := String(Info.szDisplayName)
  else
    Result := FileName;
end;
function ExtractLongPathName(const PathName: String): String;
var LastSlash, PathPtr: PChar;
begin
  Result := '';
  PathPtr := PChar(PathName);
  LastSlash := StrRScan(PathPtr, '\');
  while LastSlash <> nil do begin
    Result := '\' + ExtractLongFileName(PathPtr) + Result;
    if LastSlash <> nil then begin
      LastSlash^ := #0;
      LastSlash := StrRScan(PathPtr, '\');
    end
  end;
  Result := PathPtr + Result;
end;
```

Unfortunately, this is not so when going the other way.

It's not so bad if you simply want a file name or a directory/folder name converted from short to long format, but a full path to a file requires dealing with bit by bit. So the program uses two routines, the aforementioned `ExtractLongPathName` and a helper routine called `ExtractLongFileName`. An individual file or folder name can be converted by locating it with the Delphi `FindFirst` routine. Pass it a short name and the `Name` field of its `TSearchRec` record gets set to the long version.

In fact, the `TSearchRec` record has another record as one of its fields. The underlying Win32 `TWin32FindData` record is stored as the `FindData` field. This record has two fields of its own called `cFileName` and `cAlternateName`. These character arrays contain the long and alternate (that is, short) versions of the name, in case you need to use them in this format.

Listing 5 shows `ExtractLongFileName` using this information to endeavour to translate from short to long names. Notice that if the file cannot be found, or if it appears to have no long name equivalent, the short name is returned. This is based upon code I first saw posted on CIX by John Atkins.

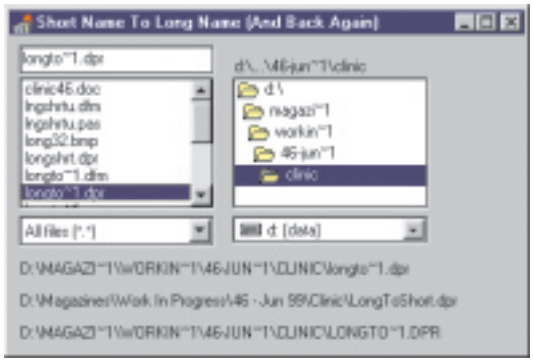
`ExtractLongPathName` takes a full path to a file and works from the file name, all through the directory names, building up a long equivalent (see Listing 6). This is also based on some code spotted on CIX, posted by Nic Landmark.

So now the program is complete. But, as per usual in Delphi-land, there are other ways of doing things. LongToShort2.Dpr does the same job, but with different implementations of these two short to long conversion routines. `ExtractLongFileName` uses a shell routine to do the hard work, and `ExtractLongPathName` uses `PChar` manipulation to produce shorter code (based on code posted to the Usenet by Bryan Wilken). Listing 7 shows these alternative versions.

So that's Win32 dealt with. Now, what about 16-bit applications written in Delphi 1? If they need to convert short to long filenames and back again, things are a little trickier. Applications running under Windows 95/98 are fine, since there is a special interrupt that will do both conversions. But if running on NT..., well I think options are rather remote.

There is a Delphi 1 project supplied called LongShrt.Dpr that again duplicates the previous projects. The only difference is that since 16-bit applications deal in short filenames by default, the program goes from short to long and back again (see Figure 3). Listing 8 shows how the code looks. The interrupt is called through inline assembler, where the `CL` register is set to 1 or 2, depending which way the translation is supposed to go.

➤ *Figure 3*




➤ *Listing 8*

```
procedure ExtractOtherNameC(SrcName, DestName: PChar; ShortToLong: ByteBool);
  assembler;
asm
  lds si, SrcName
  les di, DestName
  mov ax, $7160
  mov cl, ShortToLong
  inc cl { 1 - get short name, 2 - get long name }
  mov ch, 0 { return full path }
  int $21
end;
function ExtractShortPathName(const FileName: string): string;
var LongNameBuf, ShortNameBuf: array[0..255] of Char;
begin
  ExtractOtherNameC(StrPCopy(LongNameBuf, FileName), ShortNameBuf, False);
  Result := StrPas(ShortNameBuf)
end;
function ExtractLongPathName(const FileName: string): string;
var LongNameBuf, ShortNameBuf: array[0..255] of Char;
begin
  ExtractOtherNameC(StrPCopy(ShortNameBuf, FileName), LongNameBuf, True);
  Result := StrPas(LongNameBuf)
end;
```