Under Construction: Component Help

by Bob Swart

In the previous columns, we have examined the process of writing our own visual and non-visual components, optionally based on a DLL engine, and adding a custom bitmap. This time, we'll focus on the final touch of Component Building: the integrated Help file.

First of all, let's just build a small example component with some properties and methods. This time, I'd like to write yet another DLL wrapper, around the UUCODE.DLL. The UUCode dynamic link library implements the uuencode and uudecode file conversion algorithms that can be used to transfer files on the internet (previously used in unix-to-unix file transfers). The objective of uuencoding is to encode a file which may contain any (binary) characters into another file with a standard character set ('!"#$%&'()*+,-./012356789:;<=>?@ABC...XYZ[\]^_) that can be sent reliably over diverse networks.

Import Unit

Before we start, we first need to write an explicit import unit for the UUCODE.DLL. It's important to write an explicit import unit and not an implicit one, because we would like to use the remaining part of COMPLIB.DCL even if our DLL is not available. Listing 1 shows the import unit.

TUUCode Component

The component itself has two published properties (`inputfile` and `outputfile`), two hidden private fields (`finputfilename` and `foutputfilename` to hold the property values), two public methods (`uuencode` and `uudecode`) and one public constructor to check and see if the UUCODE.DLL could be loaded (otherwise the construction of the component will fail with a `EUUCode` exception "UUCode.DLL not loaded." Finally, we have two additional protected methods called `InputFilePChar` and `OutputFilePChar` that are needed since the `TUUCode` component works internally with `String` types to store the input and output filenames, while the UUCODE.DLL is expecting Windows-style null-terminated `PChar` types. No big deal and only visible to the component builder anyway. The component is shown in Listing 2.

Casualties

I'm sure that if you've read this column before the code in Listing 2 will hold few surprises for you; it's just another component wrapper based on an explicit import unit. But what about for the casual Delphi user? How will the component user (rather than builder) react if s/he sees the two published properties? Will s/he in fact know what `TUUCode` actually does? It takes an `inputfile` and an `outputfile`, but what is that darn `uuencode` algorithm? Some kind of cryptography perhaps?

Personally, whenever I'm not sure about something, I always hit the F1 key to get help on the component itself or one of its properties. Only, since this is but a "third-party" custom component,

➤ *Listing 1 Import unit for UUCODE.DLL*

```
{$A+,B-,D-,F-,G+,I+,K+,L-,N+,P+,Q-,R-,S+,T+,V-,W-,X+,Y-}
unit UUCode;  { NB for usage notes see file on disk }
interface
Type
  TCallBack = procedure (Position, Size: LongInt); { export; }
Const
  UUCodeLoaded: Boolean = False; { presume nothing! }
var
  UUEncode: function(FileName: PChar): Word;
  UUDecode: function(FileName: PChar): Word;
  UUEncoder: function(InFile,OutFile: PChar; Flag: Word; Unix: Boolean;
  CallBack: TCallBack): Word;
  UUDecoder: function(FileName: PChar; CallBack: TCallBack): Word;
implementation
{$IFDEF WINDOWS}
uses WinProcs;
Const SEM_NoOpenFileErrorBox = $8000;
{$ELSE}
uses WinAPI;
{$ENDIF}
var
  SaveExit: pointer;
  DLLHandle: Word;
procedure NewExit; far;
  begin
    ExitProc := SaveExit;
    FreeLibrary(DLLHandle)
  end {NewExit};
begin
  {$IFDEF WINDOWS}
  SetErrorMode(SEM_NoOpenFileErrorBox);
  {$ENDIF}
  DLLHandle := LoadLibrary('UUCODE.DLL');
  if DLLHandle >= 32 then begin
    UUCodeLoaded := True;
    SaveExit := ExitProc;
    ExitProc := @NewExit;
    @UUEncode := GetProcAddress(DLLHandle,'UUENCODE');
    @UUDecode := GetProcAddress(DLLHandle,'UUDECODE');
    @UUEncoder := GetProcAddress(DLLHandle,'UUENCODER');
    @UUDecoder := GetProcAddress(DLLHandle,'UUDECODER')
  end
end.
```

Delphi cannot just make up some help for us and instead responds with the bleak message “*There is no context-sensitive Help registered for this topic*”.

OK, so this may just be the end of the usage period of my TUUCode component, right? Wrong! Delphi is an open enough environment to let us even install our own Component Help! And that’s what we’ll be doing in the rest of this article.

WinHelp

First of all, we need to create a WinHelp skeleton file (using a WinHelp Authoring tool, such as ForeHelp for example) with only eight topics (six real topics and two popup pages):

- TUUCode (main)
- constructor Create
- methods (popup page)
- procedure UUEncode (method)
- procedure UUDecode (method)
- properties (popup page)
- property InputFile
- property OutputFile

Actually only three of them are really needed, namely the TUUCode main topic and the two property topics. The other topics are only to make the help support somewhat more complete.

I used ForeHelp 1.04 to generate the help file skeleton. A prime page for the TUUCode component with two popup pages (properties and methods) and one jump to the constructor. The two popup pages would each have two entries (InputFile and OutputFile, and UUEnCode and UUDeCode respectively). This way, the way the TUUCode component’s help works is comparable to the general Delphi help files (just drop a TEdit component and hit F1 to see the general Component Help outline of Delphi itself). Figure 1 shows the outline (in ForeHelp’s Grapher utility).

After I generated the help skeleton, I saved the project and edited the .RTF file with WinWord 2.0c (any RTF-editor will do). Using WinWord, I could enter the contents of the eight topics, and more important, I could add the special “B”-keywords that Delphi needs in order to make your help file really integrated with the

➤ *Listing 2 TUUCode Component*

```
unit TBUUCode;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, UUCode;
type
  EUUCode = class(Exception);
  TUUCode = class(TComponent)
  public    { Public class declarations (override) }
    constructor Create(AOwner: TComponent); override;
  private   { Private field declarations }
    FInputFileName: String;
    FOutputFileName: String;
  protected { Protected method declarations }
    function InputFilePChar: PChar;
    function OutputFilePChar: PChar;
  public    { Public interface declarations }
    procedure UUEncode;
    procedure UUDecode;
  published { Published design declarations }
    property InputFile:  String read FInputFileName write FInputFileName;
    property OutputFile: String read FInputFileName write FInputFileName;
  end;
procedure Register;

implementation

constructor TUUCode.Create(AOwner: TComponent);
begin
  if not UUCodeLoaded then raise EUUCode.Create('UUCode.DLL not loaded');
  inherited Create(AOwner);
end {Create};

function TUUCode.InputFilePChar: PChar;
begin
  FInputFileName[Length(FInputFileName)+1] := #0;
  InputFilePChar := @FInputFileName
end {InputFilePChar};

function TUUCode.OutputFilePChar: PChar;
begin
  FOutputFileName[Length(FOutputFileName)+1] := #0;
  OutputFilePChar := @FOutputFileName
end {OutputFilePChar};

procedure TUUCode.UUEncode;
var Error: Word;
begin
  if FInputFileName = '' then
    raise EUUCode.Create('InputFileName is empty');
  if FOutputFileName = '' then
    raise EUUCode.Create('OutputFileName is empty');
  Error := UUEncoder(InputFilePChar,OutputFilePChar,664,False,nil);
  case Error of
    1: raise EUUCode.Create('UUEnCode: input file is output file');
    2: raise EUUCode.Create('UUEnCode: input file does not exist');
    3: raise EUUCode.Create('UUEnCode: output file exists');
    4: raise EUUCode.Create('UUEnCode: could not create output file');
    5: raise EUUCode.Create(
         'UUEnCode: DLL busy, try again later (shared buffers)')
  { else OK }
  end
end {UUEncode};

procedure TUUCode.UUDecode;
var
  Error: Word;
begin
  if FInputFileName = '' then
    raise EUUCode.Create('InputFileName is empty');
  Error := UUDecoder(InputFilePChar,nil);
  case Error of
    1: raise EUUCode.Create('UUDECode: input file is output file');
    2: raise EUUCode.Create('UUDECode: input file does not exist');
    3: raise EUUCode.Create('UUDECode: output file exists');
    4: raise EUUCode.Create('UUDeCode: could not create output file');
    5: raise EUUCode.Create(
         'UUDeCode: DLL busy, try again later (shared buffers)')
  { else OK }
  end
end {UUDecode};

procedure Register;
begin
  RegisterComponents('Dr.Bob', [TUUCode]);
end {Register};
end.
```

Delphi help files (ie footnotes which have "B" as the custom mark, as shown in the dialog from WinWord in Figure 2; the footnote panel, showing the footnote text, is underneath the dialog).

The "B"-footnotes are only needed for the three topics that actually integrate with Delphi, which are the `TUUCode` main page and the two properties (events are also on this list, but we have no events for the `TUUCode` component).

The TUUCode main topic page needs a "B"-footnote that says `class_TUUCode`, ie the class name with `class_` as prefix. Also, the property topics need "B"-footnotes with the name of the property and the `prop_` prefix. For class-specific properties, we need to include the class name as well, for example `prop_TUUCodeInputFile`.

Keywords

After we've added the three "B"-footnotes to the help file (and even before we've actually written the contents of the help file), we can generate the keywords from this file that are needed to integrate with the Delphi multihelp environment.

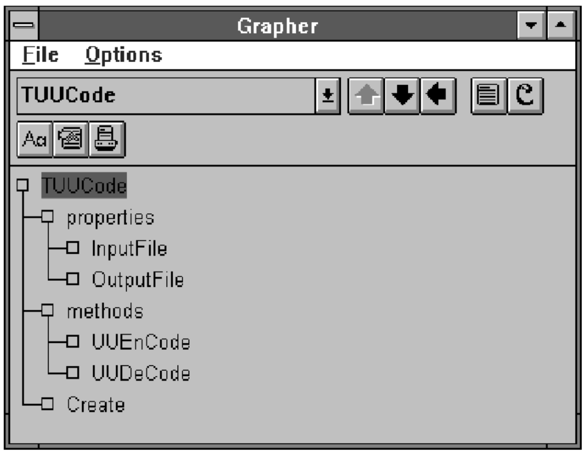
At this point, Delphi's own *Component Writer's Guide* seems to be a little out of date. First of all, the KWGEN application is a Windows application and not a DOS one any more. Second, we don't need to put the keyword file in the same directory as our compiled unit and help file, as we'll see shortly.

KWGEN allows us to browse for any .HPJ file. It then opens the corresponding .RTF file and scans (among others) for the "B"-footnotes to generate a special .KWF keyword file (see Figure 3).

Installation

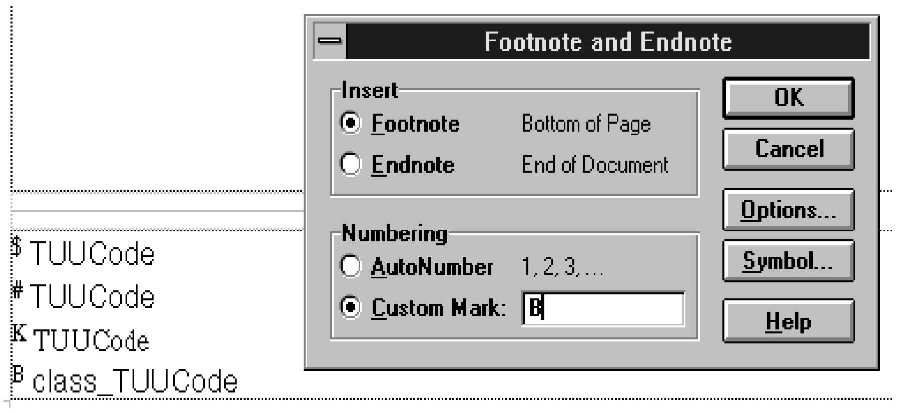
As I've said before, we don't need to put the generated .KWF file next to the compiled unit and help file. Instead, we seem to be forced to place this file in the `\DELPHI\HELP` directory where the other Delphi .KWF files can be found. Now, we can use HELPINST.EXE to generate the Delphi MultiHelp master index DELPHI.HDX file which contains references to all .KWF files in the list. *Remember to close Delphi before you attempt to do this, and to make a backup of the DELPHI.HDX file* (so if something goes wrong, you can always restore your master index). See Figure 4. The HELPINST program will start in the `DELPHI\HELP` directory, by the way, while the DELPHI.HDX file resides in the `\DELPHI\BIN` directory. Just to let you know...

I found that if I place the file TBUUCODE.KWF anywhere else, then the second time I fire up HELPINST it will say that the file TBUUCODE.KWF was not found, so



```
TUUCode
  properties
    InputFile
    OutputFile
  methods
    UUEnCode
    UUDeCode
  Create
```

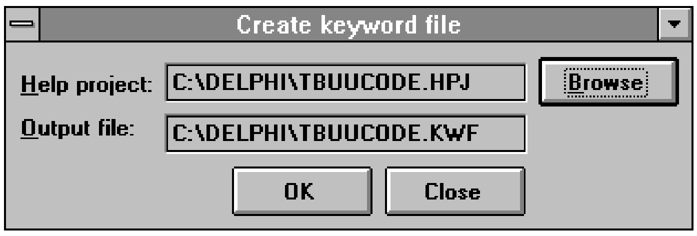
➤ Figure 1 Outline of the TUUCode component help



(● Footnote) Bottom of Page
( ) Endnote End of Document
( ) AutoNumber 1, 2, 3, ...
(● Custom Mark:) B

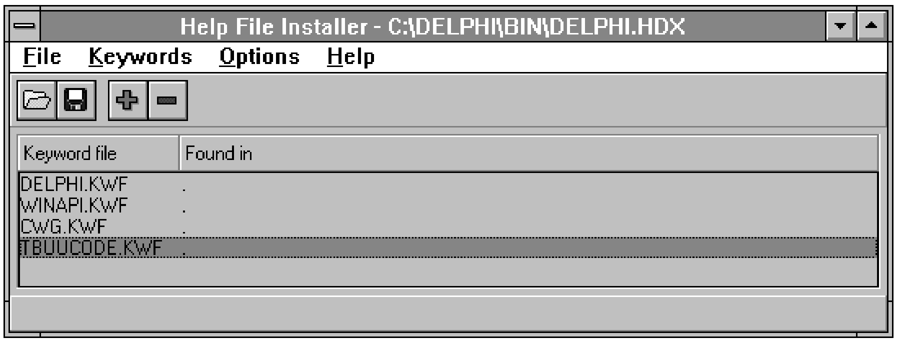
$ TUUCode
# TUUCode
K TUUCode
B class_TUUCode

➤ Figure 2 Entering "B"-Footnotes in Word for Windows



Help project: C:\DELPHI\TBUUCODE.HPJ
Output file: C:\DELPHI\TBUUCODE.KWF

➤ Figure 3 Using KWGEN



Help File Installer - C:\DELPHI\BIN\DELPHI.HDX

Keyword file	Found in
DELPHI.KWF	.
WINAPI.KWF	.
CWG.KWF	.
TBUUCODE.KWF	.

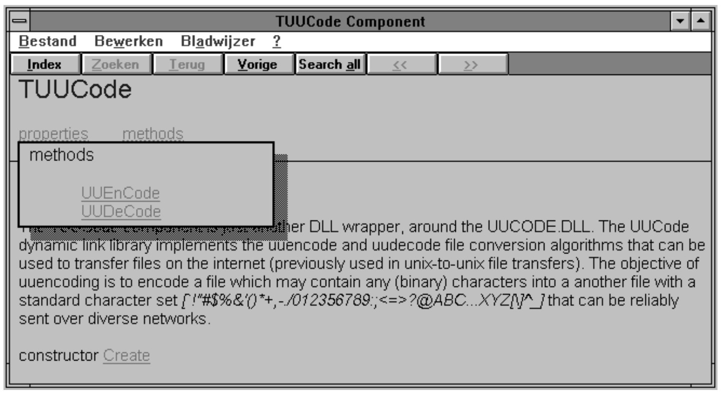
➤ Figure 4 Delphi's HELPINST utility

I had to enter it again. I overcame this problem by placing all .KWF files in the same directory.

There's a similar story for the TBUUCODE.HLP file. We find the *Component Writer's Guide* recommends placing this help file in the same directory as the compiled component unit, but I found that the easiest (and default) method was to place it in the `\DELPHI\BIN` directory. If we want to place it anywhere else, we need to modify the WINHELP.INI file.

Component Help

Well, now that we've generated a keyword file and integrated it into the Delphi MultiHelp master index, all we need to do is to finish our help file itself, then place it in the correct directory (`\DELPHI\BIN` by default). Then we can activate the `TUUCode` component help in several ways (provided we've also installed the component): drop a `TUUCode` component on a form and press F1 (`TUUCode` main help page), or, go to the Object Inspector on property `InputFile` and press F1 (property help). Figure 5 shows the final help file.



TUUCode

methods

UUEnCode
UUDeCode

[illegible]ther DLL wrapper, around the UUCODE.DLL. The UUCode dynamic link library implements the uuencode and uudecode file conversion algorithms that can be used to transfer files on the internet (previously used in unix-to-unix file transfers). The objective of uuencoding is to encode a file which may contain any (binary) characters into a another file with a standard character set *[`!"#$%&'()*+,-./012356789:;<=>?@ABC...XYZ[\]^_]* that can be reliably sent over diverse networks.

constructor Create

➤ *Figure 5 The help file in action*

Finishing Touch

If we want to search for topics in the help file we've just made, we have to make sure we've also included regular "K"-footnotes (the keywords to the topics), so we can even search in the help file by using the `Search All` button and typing `TUUCode`, for example.

Next Time

So, you think now you've seen it all when it comes to component building? Not quite. Next time we'll do something we've not done before (and I mentioned it earlier): adding custom events to our components. What's that: custom events? How? What? Yes, that's exactly what we're gonna find out next time. Stay tuned!

Bob Swart is a professional 16- and 32-bit software developer using Borland Pascal, C++ and Delphi. In his spare time, he likes to watch video tapes of Star Trek Voyager with his 1.5 year old son Erik Mark Pascal. Email Bob on 100434.2072@compuserve.com