Tips & Tricks

Thunk Update

Last month's issue featured a fairly hefty article on calling 16-bit code from applications running in Windows 95 (NT doesn't support doing this). One important thing I missed was to mention that Roy Nelson of Borland's European Technical Team was responsible for a lot of the groundwork for the code in that article – many thanks Roy!

The article suggested you write (small) snippets of code to do the actual job of calling the target routines. If assembler worries you, then fear not. Since writing the article I've been doing some homework and have come up with a way of getting the same effect without writing any assembler at all. It all revolves around a routine that does all the assembler for you.

It works out the details of all the parameters and calling conventions, including the troublesome pointers. The only problem with `Call16BitRoutine` is that it will be slower than doing a direct assembler routine. This is due to the amount of code required to take account of all the relevant possibilities: it stretches to over 100 lines of code. If efficiency is not a major factor, this function may be a viable option.

`Call16BitRoutine` can handle routines that take any 1, 2 or 4 byte ordinal parameters and also any pointer parameters (which caters for routines that take `var` and structured `const` parameters). It also deals with any function return value that is a 1, 2 or 4 byte ordinal, or a pointer. The function's declaration looks like this:

```
function Call16BitRoutine(
  Name: String; DllHandle: THandle16;
  Convention: TConvention; Args: array of const;
  ArgSizes: array of Integer): Longint;
```

The idea is that you are still responsible for loading and freeing the DLL with `LoadLibrary16` or `LoadLib16` and `FreeLibrary16` as discussed before, and you pass in the routine name and the DLL handle. The example project used last time was QTTEST.DPR. This month, a new version, QTTEST2.DPR, is supplied on the disk that uses the routine. The program loads the 16-bit DLL in one of its units' initialisation sections and frees it in the corresponding finalisation section. Note that you have to be careful doing this: remember that the platform check discussed last month happens in one of the other units' initialisation sections. It is important to ensure the platform check occurs before the library is loaded. Some sensibly placed breakpoints can verify things are working (or not).

To specify the calling convention you also pass a value of `ccPascal` or `ccCDecl` (values defined in an enumerated type called `TConvention`). The parameters are passed in as an open array. For non-pointers, you pass the relevant value. For `var` and structured `const` parameters, pass the address of the variable. For pointer parameters, pass the appropriate pointer value. In order for `Call16BitRoutine` to push the correct number of bytes onto the stack, you also need to specify how many bytes each parameter takes. Without this, the implicit type promotion would cause havoc. In the case of non-pointers, pass the number of bytes the parameter is defined to take up. For pointers (which are all four bytes in size), the number you specify should be the number of data bytes the pointer points to.

If no parameters are required, pass a zero in the argument array, and also a zero in the argument size array.

The function return value is a `Longint`, so it may need typecasting to an appropriate type, or you may need to read the low word or perhaps just the low byte. QTTEST2.DPR calls all the same 16-bit DLL routines as QTTEST.DPR (supplied as well, for comparison) but using `Call16BitRoutine`. Listing 1 shows a few of the button `OnClick` handlers from the project: the parameter-less Pascal procedure, the two parameter Pascal procedure, the two parameter C function and the function that takes a `PChar` and returns a `PChar`. Refer back to Issue 12 for details of the 16-bit DLL routines.

When using `Call16BitRoutine`, here are a few salient points to bear in mind:

1. When passing a `PChar`, or string pointer, remember that its size will be one more than its length (to take into account the terminating zero byte, or length byte respectively).

2. If the function returns a pointer, remember it will be a 16-bit pointer. Pass it to `Ptr16To32Fix` before reading from it, and to `Ptr16To32Unfix` afterwards.

3. To make the availability of the 16-bit DLL as extensive as possible, QTTEST2.DPR loads it and unloads it

➤ *Listing 1*

```
Call16BitRoutine(
  'NoParameters',DllHandle,ccPascal,[0],[0]);
...
Call16BitRoutine(
  'Proc2ParamsPascal', DllHandle, ccPascal,
  [5, 20], [SizeOf(Longint), SizeOf(Longint)]);
...
ShowMessage(Format('Sum of parameters = %d',
  [Call16BitRoutine('Func2ParamsC', DllHandle,
  ccCdecl, [5, 20], [SizeOf(Longint),
  SizeOf(Longint)])]));
...
var
  ReturnedMsg: PChar;
const
  Msg: PChar = '32-bit call';
...
ReturnedMsg := PChar(Call16BitRoutine(
  'FuncPointerParam', DllHandle, ccPascal, [Msg],
   [Succ(StrLen(Msg))]));
ShowMessage(Format('Msg received from 16-bit: %s',
  [StrPas(Ptr16To32Fix(ReturnedMsg))]));
Ptr16To32Unfix(ReturnedMsg);
```

in a unit initialisation and finalisation section. If you do likewise you need to be careful about the order of execution of initialisation sections. If this call to LoadLibrary16 occurs before the QTThunkU unit initialisation section's platform check, then your program may terminate with an unpleasant OS error, instead of the intended exception message (if running under NT).

4. When dealing with PChars returned from 16-bit DLLs, translate them into String types before passing them to the Format family of routines (as done in Listing 1). If you leave them as PChars, for some reason Windows 95 can experience certain problems.

Contributed by Brian Long

Automatic Initialisation And Finalisation

I recently found myself writing a unit for re-use by other programmers that required initialisation before any calls could be made to the unit, and clean-up (freeing resources etc) after they had finished with it. There is an easy way to make this automatic, with no coding required by the user programmers. In Delphi 2, you can put code in the initialization and finalization sections (note the strange American spelling) as shown in Listing 2. The initialization code gets executed as soon as the unit is created, and the finalization code as it is destroyed. However, Delphi 1 doesn't have a finalization section. One possible solution is to call the SysUtils routine AddExitProc, as shown in Listing 3. This adds the given procedure to the run-time library's exit procedure list, so when an application terminates its exit procedures are executed in reverse order of definition. You can pass any procedure (not function) as its parameter, but it cannot have any parameters and must be declared far.

Contributed by Jim Cooper, CompuServe 101641,440

Easy Word Wrapping

Ever wanted to wrap text in a grid cell or component that doesn't normally support it? Worried about calculating text size, which words to wrap at and all that stuff? Well, there is an easier way: use the Windows API routine DrawText. The Delphi help contains information on all the flags available, but the important one to note is DT_WORDBREAK. Use or combine any of the flags you require. Listing 4 shows an example of a TStringGrid OnDrawCell event handler that will wrap text. The grid has DefaultDrawing set to True, by the way. You could easily build this behaviour into a descendant component, of course. Remember to put WinTypes and WinProcs in a uses clause, if they aren't there already.

Contributed by Jim Cooper, CompuServe 101641,440

Many thanks to those who have contributed Tips for this column and are waiting patiently to see them appear in print! We've been tight for space recently so haven't been able to include as many as we'd like to, but will do our best to expand Tips & Tricks in subsequent issues, so please do keep them coming in!

➤ *Listing 2*

```
unit Unit1;
interface
{ Your code }
implementation
{ more code }
procedure MyInitProc;
begin
end;
procedure MyCleanupProc;
begin
end;
initialization
  { Any amount of code can go here, but calling a
    procedure looks neater, eg: }
  MyInitProc;
finalization
  { Any amount of code can go here too, but calling
    a procedure looks neater, eg: }
  MyCleanupProc;
end.
```

➤ *Listing 3*

```
unit Unit1;
interface
{ Your code }
implementation
uses  SysUtils;  { You need this }
{ more code }
procedure MyInitProc;
begin
end;
procedure MyCleanupProc; far;
  { This must be declared far }
begin
end;
initialization
  { Any amount of code can go here, but calling a
    procedure looks neater, eg: }
  MyInitProc;
  AddExitProc(MyCleanupProc);
end.
```

➤ *Listing 4*

```
procedure TForm1.MyGridDrawCell(
  Sender: TObject; Col, Row: Longint; Rect: TRect;
  State: TGridDrawState);
var
  TextStr : array[0..255] of Char;
begin
  { Because DefaultDrawing is set to True, the cell
    will actually be drawn once, so call FillRect to
    blank it out. This may seem like more work, but in
    fact everything like setting the font, pen and
    brushes is set this way, so all the text drawing
    is simpler as none of this behaviour needs to be
    reproduced here. }
  MyGrid.Canvas.FillRect(Rect);
  StrPCopy(TextStr, MyGrid.Cells[Col, Row]);
  Rect.Top := Rect.Top+2;  {Adjust for looks}
  DrawText(MyGrid.Canvas.Handle, TextStr,
           StrLen(TextStr), Rect,
           DT_CENTER or DT_NOPREFIX or DT_WORDBREAK);
end;
```