Beating The System: Thunking For Profit And Pleasure...

by Dave Jewell

Of the various development systems that I use on a regular basis, Delphi makes it easiest to write code that's readily portable between 16- and 32-bit platforms. This isn't only true of application development, it's often true when you're developing components too. This two way portability is a testimony to the excellent VCL and the way in which it hides the details of the underlying Windows API.

However, life being what it is, there will often be times when you're saddled with a large amount of "legacy" code which wasn't written with Delphi. You may not even have access to the source code – if you're coming from a traditional C/C++ development background, you may be used to working with third-party DLLs such as communications libraries, statistical packages and so forth. If the vendors of these packages have gone out of business, or don't have plans to port them to 32-bits, then you're stuffed – or are you?

Wouldn't it be great if you could call your old 16-bit DLLs from a 32-bit Delphi 2 application? Well, thunking will allow you to do exactly that. Thunking can also be used to make 32-bit API calls from a 16-bit application. This is particularly useful if you want to make use of some of those juicy new Win32 features but you aren't quite ready to make the move to 32-bits yet. With careful coding, it's possible to come up with an application which will run under both Windows 3.1 and 32-bit platforms, using certain 32-bit features if they're available. This is especially relevant when designing installation packages: under Windows 95 they should ideally be able to install desktop shortcuts, add options to the start menu and so forth.

	Windows NT	Windows 95	Win32s
Universal Thunk	No	No	Yes
Flat Thunk	No	Yes	No
Generic Thunk	Yes	Yes	No

➤ *Table 1: Thunk Flavours*

Different Flavours Of Thunks

OK, so hopefully you're sold on the idea of thunking. Now let's take a look at the various options available. There are actually three different flavours of thunks available as summarised in Table 1.

It's amusing to note that the so-called *Universal Thunk* is extremely non-universal in practice! It's only applicable if you're programming specifically for the Win32s platform. You may remember that Win32s is a collection of DLLs and VxDs which allow 32-bit applications to execute under Windows 3.1. As with other Win32s issues, there are a number of caveats relating to the use of universal thunks and because of their specialised nature I won't be discussing this technique any further.

Of much more interest is the *Generic Thunk*. Generic thunks are supported under both Windows NT and Windows 95 – that's the good news. The bad news is that generic thunks will only allow you to call 32-bit code from a 16-bit application. You can't do things the other way round. However, because generic thunks work under both NT and Windows 95, they represent a good design choice if (as noted earlier) you're in the business of developing an installation package or you want to use certain 32-bit API calls in your program without porting everything across to 32-bits.

The final thunking technique is the so-called *Flat Thunk*. Again, there's good news and bad news. On the positive side, flat thunks will allow you to call 32-bit code from 16-bit applications *and* vice versa. On the negative side, flat thunks are only supported under Windows 95 at the present time and you need to make use of Microsoft's thunk compiler.

To summarise, then, your choice of thunking technique depends on what's most important to you. If your main concern is source code portability between Windows NT and Windows 95, then Microsoft recommend that you use generic thunks. On the other hand, if the most important thing is to be able to call existing 16-bit code from a 32-bit application, and you're not too fussed about NT, then use flat thunks. Even discounting universal thunks, there is a third alternative which can be used to get you out of trouble in Windows 95 only situations. This employs a little-known Kernel routine called `QT_Thunk`. `QT_Thunk` will be covered in an article next month by Brian Long.

New Name, Same Face!

From this issue, we've decided to re-name Dave Jewell's regular column, as you will see, to reflect the fact that he's dealing with operating system and API related issues, rather than the innards of Delphi itself. As Dave shares more of his hard-won experience month by month, we hope that you will find that in your own applications you can indeed ***beat the system!***

The Generic Thunk Approach

Generic thunks are based on a relatively small number of API calls, all implemented within the Windows 95/NT Kernel. Despite the fact that these routines are documented, Borland do not provide function prototypes so I've rolled my own as shown in Listing 1.

The `LoadLibraryEx32W` routine corresponds to the usual call to the `LoadLibrary` API. From your 16-bit application, you pass it a pointer to the required 32-bit DLL and it gives you back a 32-bit module handle for the library. The `hFile` parameter is reserved for future use and must always be zero. If you look up the routine name on your MSDN CD-ROM, you'll find that there are some extra flags you can pass via the third parameter which aren't required under normal circumstances. Incidentally, unlike its 16-bit counterpart, this routine seems to always return zero in the case of failure and a non-zero value if it worked. This is just as well since, conventionally, you check for success by testing the `LoadLibrary` return value to see if it's greater than or equal to `$20`. This won't work with `LoadLibraryEx32W` because 32-bit module handles have bit 32 set, which means that the compiler will interpret a valid result as a negative number! Remember that there is no unsigned 32-bit data type under 16-bit Delphi.

The `FreeLibraryEx32W` routines forms the 32-bit counterpart to the familiar `FreeLibrary` call. You pass the 32-bit module handle to this routine when you've finished with the library. `GetProcAddress32W` takes a module handle and a procedure name, returning the address of the DLL routine you want to call. This is where you need to tread carefully! Bear in mind that what you've been given is a 32-bit linear procedure address, not a "16:16" format (segment:offset) which is what you will be used to dealing with. You can't call this address directly, since it will be meaningless to a 16-bit process and will cause an instant GPF. Moreover, you shouldn't even load this address into a segment register, since doing so will also cause a GPF, the high part of the 32-bit linear address being nonsense as far as processor is concerned. When handling what it thinks is an address, the 16-bit compiler will often load up a segment register even when you're not planning to reference that address! For example, try using the `Seg` and `Ofs` macros with an invalid address. By prototyping this routine as returning a `LongInt` instead of `TFarProc` we remove the possibility of any such embarrassment.

```
function LoadLibraryEx32W (lpszLibFile: PChar; hFile, dwFlags: LongInt):
  LongInt; far; external 'KERNEL' index 513;
function FreeLibraryEx32W (hInst: LongInt): Bool;
  far; external 'KERNEL' index 514;
function GetProcAddress32W (hInst: LongInt; lpszProc: PChar): LongInt;
  far; external 'KERNEL' index 515;
function CallProc32W (param1: LongInt; ... ; lpProcAddress32: Pointer,
  fAddressConvert, nParams: LongInt): LongInt;
  far; external 'KERNEL' index 517;
function CallProcEx32W (nParams, fAddressConvert, lpProcAddress32,
  param1: LongInt; ...): LongInt; far; external 'KERNEL' index 518;
```

➤ *Listing 1*

What is embarrassing, or at least deeply irritating, is the way in which Microsoft have defined the `CallProc32W` and `CallProcEx32W` routines. As you'll appreciate from the names, these are the routines which actually enable us to make use of the 32-bit procedure address we've retrieved. Both of these routines take a variable number of arguments and therefore won't work with Pascal (yes, I know all about `Readln` and `Writeln` but these routines are specifically 'known' to the compiler and involve all sorts of cunning behind the scenes trickery). For this reason, the last two function prototypes in the above list won't compile – they're there for illustrative purposes only.

Both routines can take up to 32 parameters. In both cases, the `nParams` field specifies how many parameters are actually being passed, while the `lpProcAddress32` field is the 32-bit procedure pointer that we got from the call to `GetProcAddress32W`. The field `fAddressConvert` is made up of a mask of 32 separate 1-bit fields, one for each possible parameter. If a particular bit is set, then the corresponding parameter is converted from a segment:offset form to a 32-bit linear address within the call. Consequently, you need to set a bit field whenever passing a pointer to a 32-bit routine. The first parameter corresponds to bit zero, the second to bit one, and so on. `CallProc32W` uses the Pascal calling convention (the routine is itself responsible for cleaning up the stack) while `CallProcEx32W` uses C calling conventions (the caller cleans up the stack).

Introducing DCallProc32

So what are we to do? Faced with a couple of routines that we can't adequately prototype but have to use, there's no choice but to bite the bullet and introduce a little assembler programming. Let's face it, if you've kept up with the disk formatting saga of the last few months, you should be able to write inline assembler in your sleep by now! However, we don't want to write reams of assembler every time we call a 32-bit API routine. In order to minimise the amount of assembler, I've written a single general purpose routine called `DCallProc32` (the D is for Delphi).

The source to this routine is shown in Listing 2. If you're familiar with the `wvsprintf` API call, you'll see that I've used the same basic approach. Rather than passing a variable number of arguments on the stack, which Delphi can't do, the routine expects a pointer, `args`, to an array of 32-bit quantities, each of which forms a parameter to the target 32-bit DLL routine we're calling. As with the other routines we've looked at, the `nParams` and `fAddressConvert` parameters specify the number of parameters and indicate which parameters should be converted to 32-bit linear

addresses. The lpProcAddress32 parameter is the procedure address of the routine we're calling.

Notice the way I've declared the CallProc32W routine. The fact that it's really a function which takes a variable number of parameters is irrelevant, this dummy declaration is enough to get the identifier into the compiler's symbol table and ensure that the relevant code is generated when the subsequent CALL instruction is executed in the in-line assembler section.

To get you started, here are a couple of examples of how you'd use the DCallProc32 routine. In the simplest case, you can pass an args pointer of Nil if the target routine doesn't take any parameters. Listing 3 gives an example of how you might call the InitCommonControls routine in the 32-bit COMCTL32.DLL library. Yes, using these techniques it's perfectly possible to build a set of routines that will allow you to use all those juicy 32-bit Windows 95 controls inside a 16-bit application and, no, I'm not going to do all the work for you! In the second example, Listing 4, we're calling the KERNAL32 library routine GlobalMemoryStatus. This routine will, amongst other things, return the total installed physical memory and the current memory load factor, information which it's normally quite difficult for a 16-bit application to determine. Notice that in this case we're passing the address of a data structure as the only parameter to the target API routine. Consequently, bit zero of the fAddressConvert parameter is set to indicate the pointer conversion is required for the first parameter.

It should be emphasised that if you use 32-bit routines extensively inside your 16-bit program, it isn't necessary to keep getting library handles and procedure addresses and then freeing the library handles. You can get the library handles for each required library just once and then free them all when the program terminates. Similarly, procedure addresses for all required routines could be retrieved just once and stored in global variables or an array of LongInt. These techniques will considerably reduce the overhead associated with each 32-bit call.

```
procedure CallProc32W; far; external 'KERNEL' index 517; { dummy declaration }
function DCallProc32(nParams, fAddressConvert, lpProcAddress32: LongInt;
  args: Pointer): LongInt; assembler;
asm
  push ds                           { stash DS register          }
  push si                           { ditto for SI register      }
  mov  cx,word ptr nParams          { number of params to push }
  jcxz @@2                          { branch if no params        }
  cld                               { ensure auto-increment      }
  lds  si,args                      { DS:SI point to arguments }
@1:
  lodsw                             { next arg (lo-order word) }
  xchg ax,bx                        { stash it in BX register  }
  lodsw                             { next arg (hi-order word) }
  push ax                           { push hi-order word         }
  push bx                           { push lo-order word         }
  loop @@1                          { loop until all done        }
  { We've pushed all the parameters, now do the control stuff  }
@2:
  push word ptr lpProcAddress32+2   { 32-bit proc addr(hi)       }
  push word ptr lpProcAddress32     { 32-bit proc addr(lo)       }
  push word ptr fAddressConvert+2   { convert mask(hi)           }
  push word ptr fAddressConvert     { convert mask(lo)           }
  push word ptr nParams+2           { hi-word of nParams         }
  push word ptr nParams             { lo-word of nParams         }
  call CallProc32W                  { let slip the dogs...       }
  pop  si                           { restore SI register        }
  pop  ds                           { restore DS register        }
end;
```

➤ *Listing 2*

```
procedure TForm1.FormCreate(Sender: TObject);
var
  hLib: LongInt;
  pInitCC: LongInt;
begin
  { Calling InitCommonControls from a 16-bit app }
  hLib := LoadLibraryEx32W('COMCTL32.DLL', 0, 0);
  if hLib <> 0 then begin
    pInitCC := GetProcAddress32W(hLib, 'InitCommonControls');
    DCallProc32(0, 0, pInitCC, Nil);
    FreeLibraryEx32W(hLib);
  end;
end;
```

➤ *Listing 3*

```
procedure TForm1.FormCreate(Sender: TObject);
var
  hLib: LongInt;
  mem: TMemoryStatus;
  pGMS: LongInt;
  pArg: Pointer; { This would be an array if more than one param }
begin
  hLib := LoadLibraryEx32W('KERNEL32.DLL', 0, 0);
  if hLib <> 0 then begin
    pGMS := GetProcAddress32W(hLib, 'GlobalMemoryStatus');
    pArg := @mem;
    mem.dwLength := sizeof(mem);
    DCallProc32(1, 1, pGMS, @pArg);
    FreeLibraryEx32W(hLib);
  end;
end;
```

➤ *Listing 4*

Other Thunking Issues

In addition to what's been described so far, there will also be times when you need to explicitly convert pointer addresses yourself, convert between 16-bit and 32-bit window handles, and so forth. Converting pointer addresses is relatively straightforward – you can do it using GetVDMPointer32W which exists in both Windows 95 and NT. This routine is defined:

```
function GetVDMPointer32W(
  lpvAddress: Pointer; fMode:
  Word): LongInt; far;
  external 'KERNEL' index 516;
```

To convert a 16-bit segment:offset pair into a 32-bit linear address, you pass it as the first parameter to the routine. The second parameter must be set to 1 if you're passing a protected mode Windows address. Set it to zero if you're passing a real-mode DOS address. The function result will give you the corresponding 32-bit linear address.

As a trivial example, the following code fragment could be used to replace the equivalent five lines in Listing 4. Here, we do the pointer conversion ourselves and therefore `fAddressConvert` is set to zero. The end result is the same.

```
pGMS := GetProcAddress32W(
  hLib, 'GlobalMemoryStatus');
pArg := Pointer(
  GetVDMPointer32W(@mem, 1));
mem.dwLength := sizeof (mem);
DCallProc32(1, 0, pGMS, @pArg);
FreeLibraryEx32W(hLib);
```

A more difficult problem concerns the issue of 16-bit versus 32-bit window handles. The way in which a 16-bit window handle is mapped onto a 32-bit handle (or vice versa) differs radically between Windows NT and Windows 95. It's therefore important to come up with a solution that is portable to both platforms. My suggestion would be to use API routines to set the state of a window, and then read back the state of that window from 'the other side' so to speak. In other words, to convert a 16-bit window handle to 32-bits, try calling `SetFocus` (for example) with the required 16-bit window handle, and then use generic thunking to call the 32-bit `GetFocus` routine to retrieve the corresponding 32-bit handle. This isn't something that I've tried – a certain amount of experimentation may be needed here to come up with something that works reliably. However, see the *Conclusions* section for information on some freeware source code which takes this further.

You also need to take care when converting segment:offset pointers to 32-bit linear addresses and passing the resulting pointer to a 32-bit call. An example of this is the code fragment above where we called `GetVDMPointer32W`. Under Windows, because of the way that the Intel segmented architecture works, a pointer to a globally locked memory block will always remain valid, even if the memory is physically moved in memory. When you use `GlobalLock` to de-reference a Windows memory handle, you get back a segment:address pair which can be used to reference the memory block. However, behind the scenes, there's an additional level of indirection which results from having the processor run in protected mode. Thus, in contrast to early versions of the Windows system, you can happily allocate a block of memory and leave it locked. The behind the scenes heap compactor will still be able to move it around as needed, and your pointer to the block will always remain valid.

But (you just knew there was going to be a 'but'!) this situation no longer applies when converting to a 32-bit linear address. Part of the raison d'être behind 32-bit programming is to get away from Intel's brain-damaged segmented architecture. The 32-bit linear address of an object won't remain

valid if the object gets moved. Consequently, it's necessary to "fix" the object in memory before making the 32-bit call. Unlike `GlobalLock`, which is more of a historical relic than anything else, `GlobalFix` really will "fix" the memory block so that it can't be moved around.

When you ask `CallProc32W` or `CallProcEx32W` to perform address translation, the Kernel will internally look at the pointer you pass, determine the corresponding global handle and call `GlobalFix` to lock the associated memory block. Under normal circumstances, you'll let the Kernel do all the address translation for you, but if you need to call `GetVDMPointer32W` to get a linear address and subsequently pass that pointer to a 32-bit routine, then you should bracket the call with `GlobalFix` and `GlobalUnfix`. If you don't know how to get a global memory handle from a pointer, take a look at the SDK documentation on `GlobalHandle`.

Conclusion

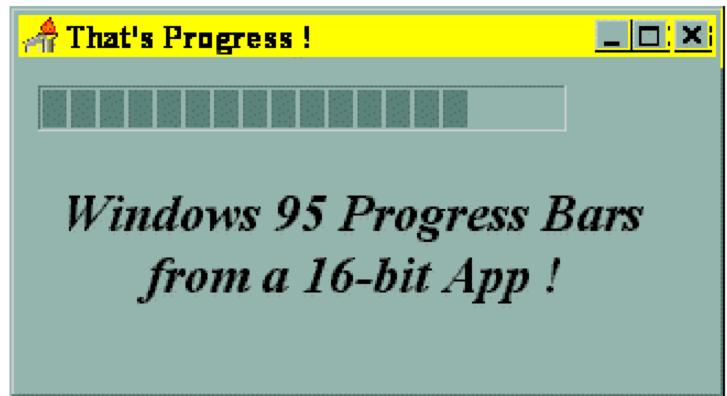
Well, that's almost all there is to say about generic thunks. Included on this month's disk is a file called CALL32NT.ZIP. This is freeware and includes complete source code for a Delphi unit which is based on the same generic thunking API described here. However, the author has taken things a step further and developed a clever system which automates the conversion of pointers and window handles when thunking up to 32-bit code. Take a look at it, you'll find it invaluable if you want to add 32-bit capabilities to your code without porting the entire thing to 32-bits.

Finally, here's an amusing little tip for those who want Windows 95 progress bars in their 16-bit apps. It turns out that the progress bar in COMCTL32.DLL can be accessed without doing any thunking at all! Just create a window of the appropriate class and use the API call `SendMessage` to get it to do the business. A small demo program is shown running in Figure 1 and the code is given in Listing 5. In actuality, this is cheating slightly: to be 'safe', you should call the 32-bit `InitCommonControls` routine to ensure that COMCTL32 has been loaded into memory – and I've already given the code to show how that's done.

In next month's issue, Brian Long will take up the story by describing flat thunks and the role of the thunk compiler and will also describe how to use `QT_Thunk`, another of Microsoft's undocumented API calls, to perform 'roll-your-own' thunking at a low-level.

I, meanwhile, shall be moving on to discuss the innards of the Windows Help file format, with some handy ideas on how you can put this knowledge to practical use.

➤ *Figure 1*




```
unit Unit1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Gauges, ExtCtrls;
type
  TForm1 = class(TForm)
    Dummy: TPanel;
    Timer1: TTimer;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
  public
    ProgressWnd: hWnd;
  end;
var Form1: TForm1;
implementation
const
  Progress_Class = 'msctls_progress32';
  PBM_SetRange    = WM_USER+1;
  PBM_SetPos      = WM_USER+2;
  PBM_DeltaPos    = WM_USER+3;
  PBM_SetStep     = WM_USER+4;
  PBM_StepIt      = WM_USER+5;
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  ProgressWnd := CreateWindow(Progress_Class, '', ws_Visible or ws_Child,
    Dummy.Left, Dummy.Top, Dummy.Width, Dummy.Height, Handle, 0, hInstance,
    Nil);
  if ProgressWnd = 0 then
    Application.Terminate;
  Dummy.Free;
end;
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  SendMessage(ProgressWnd, PBM_Stepit, 0, 0);
end;
end.
```

➤ *Listing 5*

Dave Jewell is a freelance consultant, programmer and software developer. You can contact him either as DaveJewell@msn.com or DSJewell@aol.com