Calling 16-Bit Code From 32-Bit In Windows 95

by Brian Long

When moving from Delphi 1, or indeed any 16-bit development tool, to Delphi 2, it is usually quite a large task to move a whole application in one fell swoop. It is sometimes desirable to be able to move the main executable to 32-bit, but move all the supporting files over one or two at a time. Alternatively, you may be writing a 32-bit application and want to gain access to some functionality in a 16-bit DLL that you do not have a 32-bit version of. In other words, you might want to have 32-bit code call 16-bit code. Unfortunately Win32 does not support this very elegantly.

The term used in conjunction with calling 16-bit code from 32-bit and vice versa is flat thunking. A thunk is a small nugget of code which acts as a mediator between two other pieces of essentially unrelated code. 16-bit compiled code is not capable of calling 32-bit code directly and the converse is also true. A thunk of some sort is used to set things up so that one side can call the other in a defined fashion.

There are a variety of thunking mechanisms that allow Windows code on one side of the 16-bit/32-bit fence to call code on the other side, but the required approach and terms differ for the different 32-bit platforms. See Table 1 for a matrix of the possibilities.

We'll forget about Win32s, the Win32 subset layer that can sit atop Windows 3.1x, since Delphi 2 is designed principally for Windows NT and 95. The generic thunk approach supported by Windows NT and 95 uses the same set of APIs but is restricted to 16-bit code calling 32-bit code. You should be aware that because of architectural differences between the two platforms, Microsoft advise that a generic thunk may not be entirely portable between Windows NT and Windows 95. In Windows 95, generic thunks are wrappers around flat thunks, which we will focus on here. For more details on employing generic thunks, see Dave Jewell's article last month.

Win32 platform	Thunking mechanism	Direction
Win32s	Universal thunk	32-bit EXE → 16-bit DLL
		16-bit EXE→ 32-bit DLL (unofficially)
Windows NT	Generic thunk	16-bit EXE → 32-bit DLL
Windows 95	Generic thunk	16-bit EXE → 32-bit DLL
	Flat thunk	16-bit EXE → 32-bit DLL
		32-bit EXE → 16-bit DLL

➤ *Table 1: Thunking combinations*

Flat Thunks With The Thunk Compiler

The Win32 SDK tells us that flat thunks are supported in Windows 95 by use of the Microsoft Thunk Compiler, THUNK.EXE. This is a complicated beast targeted at C/C++ programmers that takes some time to master, and usually a lot of trial and even more error each time you attempt to use it. Moreover, to successfully use it requires having the code for both the 16-bit DLL and the 32-bit EXE and re-linking both sides. This is not always feasible.

In short, flat thunking with the thunk compiler requires that you create a C-like script which the thunk compiler turns into assembler code, which needs to be assembled once to 16-bit and then again to 32-bit. The two resultant object files get linked to the 16- and 32-bit modules respectively. The 16-bit DLL source needs to be modified to include a `DllEntryPoint` routine. It also needs to import a couple of Windows 95 routines and be marked as a Windows 4.0 executable (a bit tricky in Delphi).

Many people say that if you can get a 32-bit Delphi program calling a 16-bit DLL using the thunk compiler then it will probably not end up being worth the colossal effort and heartache required. Others say flat thunking in Delphi 2 is a no-no: you must use a C or C++ product.

Because of the thunk compiler's inherent C bias, and the requirement to have both sets of source code, we will take another approach and find that Windows 95 flat thunking can be done with Delphi 2. Matt Pietrek, in *Windows 95 System Programming Secrets*, covers the basics of using an undocumented Windows 95 routine (which gets used in the code generated by the Microsoft Thunk Compiler) called `QT_Thunk`. This routine is also discussed by Andrew Schulman in *Unauthorized Windows 95* and can be used (with a bit of messing around) to call 16-bit code quite happily from a 32-bit application. In short, contrary to popular belief, flat thunks are more than possible with Delphi 2.

Cutie Flat Thunks With QT_Thunk

This cunning routine (the QT probably stands for Quick Thunk), which needs to be called from assembler, jumps off into 16-bit territory and executes a subroutine

whose address has been placed in the EDX register. In order to pass parameters to the subroutine, you need to manually push them onto the stack first. There are a number of steps to go through in order to gain success rather than misery from QT_Thunk. These will be followed up by examples which will attempt to clarify matters.

1. Reserve at least $3C (60) bytes on the stack and ensure a stack frame gets set up. This could conceivably be done by declaring a local variable at least $3C bytes in size, although that is not sufficient in Delphi 2. Because of the optimiser, this unused variable would not be compiled into the program and the required stack would not be reserved. To remedy this we can try and write a value into the variable, but again this may not be enough. Depending on the type of the variable, the optimising compiler may notice that we have written a value to a variable and not read from it, and still optimise it away. Declaring the variable as an appropriately sized string and writing to it does manage to fool the optimiser and get the desired effect.

2. Ensure an EBP stack frame gets generated, ie ensure assembler is generated in the routine's prolog code to set up the EBP base pointer register. This is sometimes done automatically, depending upon what you do in the subroutine. However, the optimiser is at liberty to not bother if it sees fit. To force a stack frame to be generated, use the {$W+} or {$StackFrames On} compiler directive before the subroutine. If an EBP stack frame is not set up, your routine will cause an *Access Violation* upon exit, as QT_Thunk will have scribbled across the calling routine's stack frame.

3. Don't declare any other stack based variables in your subroutine, since QT_Thunk has a tendency to walk over them. The purpose of the stack space reserved above is to provide a scribbling pad just below EBP for QT_Thunk to write over. Other local variables may get in the way of the EBP-relative area.

4. Don't use additional stack space in any other way for the same reason as above. For example, don't use try..except..end statements or try..finally..end statements.

5. The DLL must be loaded not with the usual LoadLibrary call, but instead with the undocumented LoadLibrary16. Since the DLL is 16-bit, LoadLibrary will fail to load it. It must similarly be unloaded with FreeLibrary16 rather than FreeLibrary.

6. The function address must be obtained not with GetProcAddress but with the undocumented GetProcAddress16. Normally when using GetProcAddress, you assign the result to a procedural variable defined with a type that looks much like the original function so that you can easily call it from Delphi code. When using QT_Thunk, there is no reason to go to this trouble, as the subroutine is called indirectly by manually pushing parameters onto the stack and jumping to QT_Thunk. A normal pointer will suffice, but defining a procedural variable will make the code more readable.

7. The parameters must be pushed onto the stack using assembler code, in the appropriate order. For routines compiled with Pascal calling conventions, this means pushing them left to right, and for C calling convention routines push them from right to left.

8. If you call a C calling convention routine, remember it is your responsibility to tidy the stack up, so increment the stack pointer, ESP, by the number of bytes collectively taken by all the routine parameters. In other words, undo all the parameter pushes you did at the start by adding an appropriate value to the stack pointer.

9. Look out for parameters of the target subroutine that are pointers, or have constituent parts that are pointers (this includes PChars). You will need to substitute in an address that means something to 16-bit code (see below). A 32-bit pointer will be meaningless to your 16-bit DLL.

10. Avoid passing object references and class references to 16-bit routines. The layout of objects and class information is rather different between the two platforms.

11. If you want to pass string information, remember to translate any Delphi 2 long strings into short strings or PChars – Delphi 1 doesn't understand these new long strings.

12. The routine's return value will be in the 16-bit registers just as it would in a 16-bit program. In other words if the routine returns a Longint, this return value will be found in DX (high word) and AX (low word). If it returns a byte, this will be in AL.

13. If a pointer value is returned from 16-bit then it will need to be transformed into the equivalent 32-bit pointer.

14. Remember that the generic types Integer and Cardinal are 16-bit values in 16-bit but 32-bit values in 32-bit. If an Integer or Cardinal variable is declared in a 16-bit DLL, use a Smallint or Word in the 32-bit EXE, or redefine Integer and Cardinal to be Smallint and Word.

15. It is wise to make a 16-bit test program that calls the target routine in the DLL before embarking on writing a 32-bit calling version. This should be loaded into Turbo Debugger for Windows and you should inspect the subroutine call in the CPU (assembler code) view. This will clarify what parameter values get pushed onto the stack and in what order. This is particularly important with non-atomic data types where there may be special values passed on the stack, eg open arrays (where a number indicating how many array elements are present is passed in addition to the array) and constant arrays.

Let's now see how all this pans out with some example code. You'll notice that all the code sections that use QT_Thunk are button event handlers. It is often desirable to wrap up a 16-bit call via QT_Thunk in a stand-alone routine. You can see an example of such a thunk function, or *thunktion*, in the later section *Finding Free System Resources*.

Note that Matt Pietrek advises that QT_Thunk should be declared as a cdecl routine. Since we call it directly from assembler, and it gets declared as a procedure with no parameters, it makes no difference

what convention it is declared with in Delphi.

Calling A Procedure

On the disk is a project called DLL16BIT.DPR which makes a 16-bit DLL. Most of its code (all of which is in the project file) is shown in Listing 1. You will need to refer to this listing often as we go through the following paragraphs, as the various routines are covered in the various sections. Amongst several other exported routines that we'll get onto later this project has a parameter-less procedure called `NoParameters`.

The 32-bit Delphi 2 project QTTEST.DPR manages to call this using the code snippet in Listing 2. The subroutine follows step 1 by declaring a string of at least $3C bytes and initialising it to a blank string. Any variables required in the subroutine are declared non-local for step 2 and the stack frame compiler directive covers step 3. The 16-bit DLL is loaded with a call to `LoadLib16`. Step 5 above advises `LoadLibrary16` and `LoadLib16` is simply a small wrapper around this call that includes some error checking (see Listing 3 for the code). The file QTTHUNKU.PAS defines this utility routine amongst others, more of which later. The DLL does get unloaded with a call to the advised `FreeLibrary16`.

Notice that this example code uses a procedural variable to store the 16-bit routine's address. This is normal practice when using explicitly loaded DLLs in Windows, but when calling a 16-bit routine from 32-bit code there is no real need for this. Since the call is being hand-coded in assembler, the type information in the procedural variable is never used and so a pointer variable would serve just as well. One plus point for the way it is written here is that the procedural variable acts as a form of self documentation: the parameters and return type are obvious by looking at the subroutine type. However, for brevity, simple pointers will be used from here on. This means that the above type declaration can be removed and the address variable can be re-declared and assigned as

```
procedure NoParameters; export;
begin
  ShowMessage('Hello world from a 16-bit DLL');
end;
procedure Proc2ParamsPascal(X, Y: Longint); export;
begin
  ShowMessage(Format('%d + %d = %d', [X, Y, X + Y]));
end;
procedure Proc2ParamsC(X, Y: Longint); cdecl; export;
begin
  Proc2ParamsPascal(X, Y);
end;
procedure ProcPointerParam(Msg: PChar); export;
begin
  ShowMessage(Format('Msg received from 32-bit: %s', [Msg]));
end;
procedure ProcVarConstParams(var Num: Smallint; const Str: String); export;
begin
  Inc(Num, 10);
  ShowMessage(Str);
end;
procedure ProcOpenArrayParam(const Numbers: array of Smallint); export;
var
  Loop: Integer;
  Sum: Longint;
begin
  Sum := 0;
  for Loop := Low(Numbers) to High(Numbers) do
    Inc(Sum, Numbers[Loop]);
  ShowMessage(Format('Sum of passed values = %d', [Sum]));
end;
function Func2ParamsPascal(X, Y: Longint): Longint; export;
begin
  ShowMessage(Format('1st: %d 2nd: %d', [X, Y]));
  Result := X + Y;
end;
function Func2ParamsC(X, Y: Longint): Longint; cdecl; export;
begin
  Result := Func2ParamsPascal(X, Y);
end;
const
  Buffer: PChar = 'Hello world, returned from 16-bit';
function FuncPointerParam(Msg: PChar): PChar; export;
begin
  ShowMessage(Format('Msg received from 32-bit: %s', [Msg]));
  Result := Buffer;
end;
```

➤ *Listing 1: Some code from the 16-bit DLL16BIT.DPR project*

```
type
  TDelphi1Proc = procedure; pascal;
var
  ProcAddress: TDelphi1Proc;
{$StackFrames On}
procedure TForm1.BtnProcNoParamsClick(Sender: TObject);
var
  EatStackSpace: String[$3C];
begin
  // Ensure buffer isn't optimised away
  EatStackSpace := '';
  // Try and load 16-bit DLL
  DLLHandle := LoadLib16('DLL16Bit.DLL');
  @ProcAddress := GetProcAddress16(DLLHandle, 'NoParameters');
  if Assigned(ProcAddress) then
    asm
      //Load routine address into EDX
      mov   edx, ProcAddress
      //Call routine
      call  QT_Thunk
    end;
  //Now release 16-bit DLL
  FreeLibrary16(DLLHandle);
end;
```

➤ *Listing 2: Calling a procedure with no parameters*

```
function LoadLibrary16(LibFileName: PAnsiChar): THandle; stdcall;
  external kernel32 index 35;
function LoadLib16(LibFileName: String): THandle;
begin
  Result := LoadLibrary16(PChar(LibFileName));
  if Result < HInstance_Error then
    raise EFOpenError.Create('LoadLibrary16 failed!');
end;
```

➤ *Listing 3: Loading a 16-bit DLL from Delphi 2*

follows:

```
var ProcAddress: Pointer;
...
ProcAddress :=
   GetProcAddress16(DLLHandle,
   'NoParameters');
```

Note that there are no parameters or return values with the `NoParameters` routine, so the remaining steps don't apply.

Passing Parameters

The DLL also has a procedure `Proc2ParamsPascal` (see Listing 1 again) that takes a couple of long integer parameters. In order to pass parameters to the subroutine via `QT_Thunk`, you need to push them onto the stack. Since this is a procedure that has been declared in Delphi 1 with no calling convention modifier, it will have been compiled using the Pascal calling convention. This means that the parameters are expected to be pushed in a left to right order, ie we need to push `X` and then push `Y`. Listing 4 shows how to achieve this.

Cdecl Routines

If the procedure is declared using the C calling convention (with `cdecl`), the parameters must be pushed in reverse order, right to left. In addition, the C calling convention requires that the function *caller* tidy up the stack, rather than the called function as is the case with the Pascal convention. This requirement means we need to increment the stack pointer by the number of bytes originally pushed onto the stack. The DLL project implements a procedure `Proc2ParamsC` which is just the same as `Proc2ParamsPascal`, but declared to be `cdecl`. The code in Listing 5 shows that a C declared routine is called practically the same as a Pascal declared one (Listing 4), but for the reversed parameter pushes and the stack increment at the end.

Passing Pointers

Here's where things start getting a bit sticky. When dealing with any pointer, you need to pass a value to the DLL that means something to it. Your 32-bit pointers are incompatible with selector/offset combinations that are used in 16-bit, so a translation process must take place. However, an additional problem is that not all memory that is addressable in a 32-bit process will be addressable from 16-bit: the potential address ranges differ. The ramification of this is that you must allocate a block of memory in a way that ensures the memory will be accessible from 16-bit DLLs. Data can then be copied into the buffer and a 16-bit version of its address can be passed to the routine. When `QT_Thunk` finishes, you need to de-allocate the memory.

GlobalAlloc16	Allocates 16-bit accessible memory with the specified flags and returns a 16-bit selector to represent it
GlobalFree16	Takes a 16-bit selector and frees the memory block it represents and presumably also frees the selector
GlobalLock16	Takes a 16-bit selector, locks it and returns a 32-bit pointer that can access the memory
GlobalUnlock16	Takes a 16-bit selector and unlocks the memory block it refers to

➤ *Table 2: Descriptions of Listing 6's routines*

```
var Param1, Param2: Longint;
...
ProcAddress := GetAddress16(DLLHandle, 'Proc2ParamsPascal');
if Assigned(ProcAddress) then begin
  Param1 := 5;
  Param2 := 20;
  asm
    push  Param1
    push  Param2
    mov   edx, ProcAddress
    call  QT_Thunk
  end;
end;
```

➤ *Listing 4: Passing parameters to a subroutine using Pascal calling conventions*

```
var  Param1, Param2: Longint;
...
push  Param2 //Note second parameter is pushed first
push  Param1
mov   edx, ProcAddress
call  QT_Thunk
//Increment the stack ptr by size of the 2 Longint parameters
add   esp, 4 * 2
...
```

➤ *Listing 5: Passing parameters to a subroutine using C calling*

```
type THandle16 = Word;
...
function  GlobalAlloc16(Flags: Integer; Bytes: Longint): THandle16; stdcall;
function  GlobalFree16(Mem: THandle16): THandle16; stdcall;
function  GlobalLock16(Mem: THandle16): Pointer; stdcall;
function  GlobalUnLock16(Mem: THandle16): WordBool; stdcall;
```

➤ *Listing 6: Undocumented 16-bit Windows 95 memory*

```
function GlobalAllocPtr16(Flags: Word; Bytes: Longint): Pointer;
begin
  Result := nil;   //Ensure memory is fixed, meaning there is no need to lock it
  Flags := Flags or gmem_Fixed;
  LongRec(Result).Hi := GlobalAlloc16(Flags, Bytes);
end;
function GlobalFreePtr16(P: Pointer): THandle16;
begin
  Result := GlobalFree16(LongRec(P).Hi);
end;
```

➤ *Listing 7: Wrapper routines to simplify 16-bit memory*

16-bit accessible memory can be managed using some undocumented Windows 95 routines. These are declared in Listing 6 and described individually in Table 2.

When attempting to use these undocumented calls from a 32-bit capable C/C++ compiler, you are forced to use specially hand-crafted .DEF files, as these routines are exported from Windows 95's Kernel32 with no names. This poses no problem for Delphi, which can link happily to the number alone. One up for Delphi developers I'm sure you'll agree.

To simplify the use of these calls, you can use some wrapper functions from QTTHUNKU.PAS to allocate and de-allocate 16-bit memory, which return and take a 16-bit pointer respectively. These are shown in Listing 7.

The allocation routine fixes the memory, meaning there is no need to lock it. Locking the memory and unlocking it in GlobalFreePtr16 is no problem, but there is a good reason for actually fixing the memory. When code is later used to translate the returned 16-bit selector/offset pointer into a 32-bit linear offset, problems are avoided by ensuring the memory is fixed. If, during the time that the 32-bit offset is being used, the Windows 95 garbage compactor kicks in, it is feasible that the 16-bit accessible memory will be moved in physical memory. This would invalidate the 32-bit pointer. Fixing the memory stops Windows physically shuffling it around.

In addition QTTHUNKU.PAS has another routine that can take a 16-bit pointer and return a 32-bit pointer to the same memory. Notice that GlobalLock16 can do this, but that routine imposes a requirement to unlock the memory afterwards using GlobalUnlock16. The Ptr16To32 routine in Listing 8 doesn't. It avoids the problem by using WOWGetVDMPointer (Windows-on-Windows Get Virtual DOS Machine Pointer).

WOWGetVDMPointer is a routine implemented in WOW32.DLL in Windows 95 and Windows NT. It can translate a 16-bit real mode or protected mode pointer into a flat 32-bit offset. Though documented, Borland have not provided import declarations for it or any other WOW routines. Ptr16To32 could just as well have been implemented using the GetThreadSelectorEntry, for which a declaration *is* present, as shown in Listing 9.

So, to pass a pointer to your 16-bit DLL, you need to allocate a new block of memory of an appropriate size, yielding a 16-bit pointer. Then you need to copy the relevant data into the buffer using a 32-bit representation of the pointer. After using the pointer you need to free it. Since allocating a 16-bit buffer and populating it are common requirements when passing pointer-based data down to 16-bit, there is another wrapper routine to do it (Listing 10). It takes a byte count and some flags, as per GlobalAlloc16, but also three more parameters. A 32-bit pointer is taken as a var parameter, which gets set to a 32-bit version of the 16-bit buffer pointer. Another var parameter, this time typeless, is used to represent the data to be copied and the last parameter dictates how many bytes will be copied.

It is safe to use Ptr16To32 on data allocated by GlobalAllocPointer16 or GlobalAllocPtr16 since they explicitly fix the allocated buffer. If you allocate memory using GlobalAlloc16 it is your responsibility to ensure the memory is fixed by using GlobalFix and GlobalUnfix on the 32-bit version of the address. The ProcPointerParam procedure from the DLL (See Listing 1) takes a pointer parameter. The pointer is a PChar and can be called using the techniques shown in Listing 11.

You'll perhaps notice that despite the flat pointer being set up, it is not being used here. However, it may need to be used after the routine has been called to get access to data that may have been modified. Later examples do this.

Const And Var Parameters

When you declare a var parameter, you are passing by reference as opposed to the default passing by value scheme. The effect is that you pass the address of the variable that was passed as the parameter, rather than a copy of its value. The same applies with

```
//Turn 16-bit pointer (selector and offset) into 32-bit pointer (offset)
function Ptr16To32(P: Pointer): Pointer;
begin
  Result := WOWGetVDMPointer(DWord(P), 0, True);
end;
```

➤ *Listing 8: How to turn a 16-bit pointer into a 32-bit pointer*

```
function Ptr16To32(P: Pointer): Pointer;
var LDTEntry: TLDTEntry;
begin
  if not GetThreadSelectorEntry(GetCurrentThread, LongRec(P).Hi, LDTEntry) then
    Result := nil
  else
    with LDTEntry do
      Result :=
        Pointer((BaseHi shl 8 + BaseMid) shl 16 + BaseLow + LongRec(P).Lo);
end;
```

➤ *Listing 9: A possible replacement for Listing 8*

```
//16-bit pointer returned. FlatPointer is 32-bit pointer
//Buffer is allocated and then DataSize bytes from Source are copied in
function GlobalAllocPointer16(Flags: Word; Bytes: Longint;
  var FlatPointer: Pointer; var Source; DataSize: Longint): Pointer;
begin
  //Allocate memory in an address range
  //that _can_ be accessed by 16-bit apps
  Result := GlobalAllocPtr16(Flags, Bytes);
  //Get 32-bit pointer to this memory
  FlatPointer := Ptr16To32(Result);
  //Copy source data into the new bimodal buffer
  Move(Source, FlatPointer^, DataSize);
end;
```

➤ *Listing 10: 16-bit memory management super wrapper routine*

structured and short string parameters passed as const parameters, with the difference that you are unable to write to a const parameter. Other const parameters are passed by value. ProcVarConstParams is a 16-bit procedure (see Listing 1) that gets called from 32-bit in Listing 12. Notice that the var and const parameters need to be placed in 16-bit accessible memory before passing the address along.

Open Array Parameters

When you pass a value to a routine declared to take an open array, ie an array argument with no specified index bounds, two values are passed. The array is passed first (if this is a var or const parameter, the address of the array is passed), and then a value indicating how many values make up the array. This value is one less than the number of array elements and is passed to 16-bit routines as a 16-bit number.

The idea is that the called routine can access the array as if its elements were numbered from zero, which would make the last index one less than the total number of elements. The functions Low and High, when applied to the array parameter in the routine's implementation, return 0 and the high index number respectively. The extra, normally hidden, parameter identifies the last array index number.

DLL16BIT.DPR has a procedure ProcOpenArrayParam that takes a const open array of Smallints (see Listing 1). Listing 13 shows the const array parameter being passed by reference, via some allocated 16-bit memory.

Function Return Values

When 16-bit functions return values, they are typically returned in various register combinations. 8-bit values are returned in AL and 16-bit values are returned in AX. Values up to 32-bit (including pointers) are returned with the low word in AX and the high word in DX. An assembler manual should describe other values such as floating point values. When a 16-bit function is called from 32-bit, the values are returned in the same register combinations.

Listing 1 shows that Proc2ParamsPascal and Proc2ParamsC (as covered earlier) have alternative definitions as functions in the DLL project: Func2ParamsPascal and Func2ParamsC. These return a calculated sum value rather than just displaying it. In Listing 14 there are two button OnClick handlers from QTTEST.DPR that call these routines and display the results obtained from the DX:AX register pair.

Pointer Return Values

If a pointer value is returned from a function, you need to turn what will be a 16-bit address into a 32-bit address. The previously described Ptr16To32 function can do this, but there is a potential pitfall.

If the 16-bit pointer refers to memory allocated by the 16-bit process, you should take an extra safety step. If the pointer refers to a buffer allocated by GlobalAllocPtr16 in your 32-bit program, the step can be avoided. The danger issue was outlined earlier: if you translate an arbitrary 16-bit address into a 32-bit linear address, the Windows system garbage collection thread can invalidate the address if the memory is not fixed. The GlobalAllocPtr16

```
var MsgBuffer, MsgBuffer16: PChar;
    Msg: PChar = '32-bit call';
...
//Get and fill 16-bit memory with source string
MsgBuffer16 := GlobalAllocPointer16(GPTR, 255, Pointer(MsgBuffer),
  (Msg^, StrLen(Msg));
asm
  push  MsgBuffer16
  mov   edx, ProcAddress
  call  QT_Thunk
end;
GlobalFreePtr16(MsgBuffer16);
```

➤ *Listing 11: Passsing a pointer from 32-bit to 16-bit*

```
var
  OldNum: Smallint;
  NumPtr, NumPtr16: ^Smallint;
  StrPtr, StrPtr16: PShortString;
  Str: ShortString = 'Hello from 32-bit';
...
OldNum := 0;
//Get and fill 16-bit buffers with source data
NumPtr16 := GlobalAllocPointer16(GPTR, SizeOf(Smallint),
  Pointer(NumPtr), OldNum, SizeOf(Smallint));
StrPtr16 := GlobalAllocPointer16(GPTR, SizeOf(ShortString),
  Pointer(StrPtr), Str, Succ(Length(Str)));
asm
  push  NumPtr16
  push  StrPtr16
  mov   edx, ProcAddress
  call  QT_Thunk
end;
ShowMessage(Format('Original var param = %d, new var param = %d',
  ([OldNum, NumPtr^]));
GlobalFreePtr16(NumPtr16);
GlobalFreePtr16(StrPtr16);
```

➤ *Listing 12: Passing const and var parameters to 16-bit*

```
type
  TNumbers = array[11..15] of Smallint;
var
  Numbers: TNumbers = (1, 2, 3, 4, 5);
  NumOfNumbers: Word;
  NumbersPtr, NumbersPtr16: ^TNumbers;
...
//Get and fill 16-bit buffer with source data
NumbersPtr16 := GlobalAllocPointer16(GPTR, SizeOf(TNumbers),
  Pointer(NumbersPtr), Numbers, SizeOf(TNumbers));
NumOfNumbers := High(TNumbers) - Low(TNumbers);
asm
  push  NumbersPtr16
  push  NumOfNumbers
  mov   edx, ProcAddress
  call  QT_Thunk
end;
GlobalFreePtr16(NumbersPtr16);
```

➤ *Listing 13: Passing an open array parameter from 32-bit*

routine ensures memory is fixed. When converting 16-bit pointers to 32-bit, where the pointer comes from the 16-bit process, use two special routines to do the job as declared in Listing 15.

These two act as replacements for the previously described WOWGetVDMPointer. The routine WOWGetVDMPointerFix does exactly the same, but also calls GlobalFix on the 32-bit pointer, and WOWGetVDMPointerUnfix calls GlobalUnfix to tidy up.

The routine FuncPointerParam from the 16-bit DLL (See Listing 1) returns a PChar. The 32-bit program passes in a PChar, translated to 16-bit, and displays the returned string using this code. Since the returned string is created in the DLL, it is only reliably usable after using the two previously described calls which have been packaged into the shorter-named Ptr16To32Fix and Ptr16To32Unfix. Listing 16 contains the code.

```
var Param1, Param2, ReturnValue: Longint;
...
ProcAddress := GetAddress16(DLLHandle, 'Func2ParamsPascal');
if Assigned(ProcAddress) then begin
  Param1 := 5;
  Param2 := 20;
  asm
    push  Param1
    push  Param2
    mov   edx, ProcAddress
    call  QT_Thunk
    mov   ReturnValue.Word.2, dx
    mov   ReturnValue.Word.0, ax
  end;
  ShowMessage(Format('Sum of parameters = %d', [ReturnValue]));
end;
...
ProcAddress := GetAddress16(DLLHandle, 'Func2ParamsC');
if Assigned(ProcAddress) then begin
  Param1 := 5;
  Param2 := 20;
  asm
    push  Param2
    push  Param1
    mov   edx, ProcAddress
    call  QT_Thunk
    //Increment the stack ptr by size of the 2 Longint parameters
    add   esp, 4 * 2
    mov   ReturnValue.Word.2, dx
    mov   ReturnValue.Word.0, ax
  end;
  ShowMessage(Format('Sum of parameters = %d', [ReturnValue]));
end;
```

➤ *Listing 14: Function return values*

```
function WOWGetVDMPointerFix(vp, dwBytes: DWord; fProtectedMode: Bool):
  Pointer; stdcall;
procedure WOWGetVDMPointerUnfix(vp: DWord); stdcall;
```

➤ *Listing 15: Extra safety in translating 16-bit pointers to 32-bit*

```
var ReturnedMsg: PChar;
...
MsgBuffer16 :=
  GlobalAllocPointer16(GPTR, 255, Pointer(MsgBuffer), Msg^, StrLen(Msg));
asm
  push  MsgBuffer16
  mov   edx, ProcAddress
  call  QT_Thunk
  mov   ReturnedMsg.Word.0, ax
  mov   ReturnedMsg.Word.2, dx
end;
ShowMessage(Format('Msg received from 16-bit: %s',
  [PChar(Ptr16To32Fix(ReturnedMsg))]));
Ptr16To32Unfix(ReturnedMsg);
GlobalFreePtr16(MsgBuffer16);
```

➤ *Listing 16: Dealing with pointer return values*

```
function User16Handle: THandle;
begin
  //Get User handle by loading it.
  Result := LoadLib16('USER.EXE');
  //Free this particular load - User will stay in memory
  FreeLibrary16(Result);
end;
```

➤ *Listing 17: Obtaining a 16-bit core system DLL module handle*

C++ DLL Routines And Linking By Number

When your 16-bit application uses DLLs written in C++, it may well avoid the issues of name mangling, or decorated C++ function names, by linking by number instead of by name. When writing a thunktion wrapper for such a routine, it may seem a little excessive to have to resort to finding the name of the routine. To avoid the problem you can take advantage of a "feature" of GetProcAddress16. You can get it to find the address of a function by specifying an ordinal number in a string format. To find the address of a routine exported with the number 45, it is fine to use:

```
ProcAddress := GetProcAddress(
  Handle, '#45');
```

Finding Free System Resources

In Windows 3.x you could find available free system resources with the GetFreeSystemResources API. For some reason, this has been removed from the Win32 API. However we can still call the 16-bit version using QT_Thunk as outlined in the previous section, and this approach is used by Matt Pietrek in his published work. GetFreeSystemResources is declared in Delphi 1's WinProcs unit and in the QTFUNCSU.PAS file on this month's disk as follows:

```
function GetFreeSystemResources(
  SysResource: Word): Word;
```

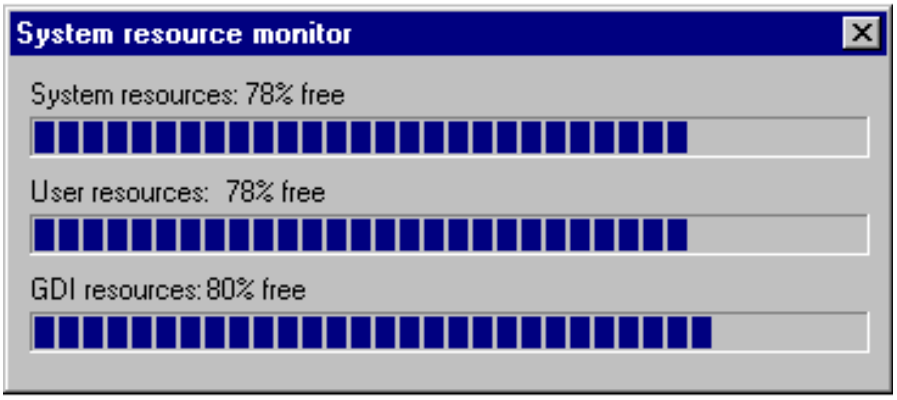
The function is implemented via a call to QT_Thunk in much the same way as shown above, but it does use another utility routine from QTTHUNKU.PAS. Since GetFreeSystemResources comes from the 16-bit Windows module USER.EXE, we can use User16Handle (see Listing 17). This, like GDI16Handle and Kernel16Handle, is a simple function that obtains the handle of a 16-bit system DLL in Windows 95.

Normally, when using a custom DLL, you load it, then get the address of a routine, call the routine, then unload the library: the loading and unloading must be discontiguous operations to ensure the DLL is present when you make the call. Since the 16-bit system DLLs are always in memory anyway, we can get one of their handles by loading the library in the usual fashion, but we can "unload" it straight away, which effectively decrements the usage counter, but leaves it in memory. This simplifies the calling code, the remains of which are in Listing 18.

This surrogate 32-bit routine, which has an identical interface to the 16-bit function it provides a wrapper for, is called a thunktion, a function implemented by a shortcut flat thunk. Having written the thunktion (in QTFUNCSU.PAS), it can be used just like it could under Windows 3.1 (it didn't exist in Windows 3.0). The project FSR.DPR has a timer on it which periodically calls `GetFreeSystemResources` and updates three progress bars with the free GDI resources, User resources and System resources. The project ends up looking somewhat like the Windows 95 resource meter application (see Figure 1).

There are a few points made in Chapter 4 of *Windows 95 System Programming Secrets* worth repeating about `GetFreeSystemResources`. Firstly, the system resource value is simply the lower of the User and GDI resource values. This point was reasonably well understood by 16-bit Windows programmers.

The second point is that in Windows 3.1, the free GDI resource value indicated the percentage of free space in the (16-bit) GDI local heap. In Windows 95, it represents the lower value of the percentage of free space in the 16-bit GDI heap and that free in the 32-bit GDI heap. It doesn't take too much guessing which one will be lower. For the User resource value, Windows 3.1 returned the lowest percentage free space in User's local heap, menu heap and string heap. In Windows 95, the heaps used are the 16-bit local heap, 32-bit window heap and 32-bit menu heap. Again,



➤ *Figure 1*

```
function GetFreeSystemResources(SysResource: Word): Word;
var EatStackSpace: array[0..$3C] of Char;
begin
  // Make sure to use the local variable so it's not optimised away
  EatStackSpace := '';
  GetFreeSystemRes := GetProcAddress16(User16Handle,(
    'GetFreeSystemResources');
  if Assigned(GetFreeSystemRes) then
    asm
      push    SysResource
      mov     edx, [GetFreeSystemRes]
      call    QT_Thunk
      mov     [Result], ax
    end;
end;
```

➤ *Listing 18: Finding free system resources with a thunktion*

```
uses QTThunkU;
{$Align Off}
{$StackFrames On}
{ Generic stuff for 16-bit }
type
  THandle = THandle16;
  Bool = WordBool;
var
  ToolHelp16Handle: Windows.THandle;
  MEPtr32Bit, MEPtr16Bit: PModuleEntry;
  TEPtr32Bit, TEPtr16Bit: PTaskEntry;
  LEPtr32Bit, LEPtr16Bit: PLocalEntry;
  SHIPtr32Bit, SHIPtr16Bit: PSysHeapInfo;
...
initialization
  MEPtr16Bit := GlobalAllocPtr16(GPTR, SizeOf(TModuleEntry));
  MEPtr32Bit := Ptr16To32(MEPtr16Bit);
  TEPtr16Bit := GlobalAllocPtr16(GPTR, SizeOf(TTaskEntry));
  TEPtr32Bit := Ptr16To32(TEPtr16Bit);
  LEPtr16Bit := GlobalAllocPtr16(GPTR, SizeOf(TLocalEntry));
  LEPtr32Bit := Ptr16To32(LEPtr16Bit);
  SHIPtr16Bit := GlobalAllocPtr16(GPTR, SizeOf(TSysHeapInfo));
  SHIPtr32Bit := Ptr16To32(SHIPtr16Bit);
  ToolHelp16Handle := LoadLibrary16('TOOLHELP.DLL');
finalization
  GlobalFreePtr16(MEPtr16Bit);
  GlobalFreePtr16(TEPtr16Bit);
  GlobalFreePtr16(LEPtr16Bit);
  GlobalFreePtr16(SHIPtr16Bit);
  FreeLibrary16(ToolHelp16Handle);
end.
```

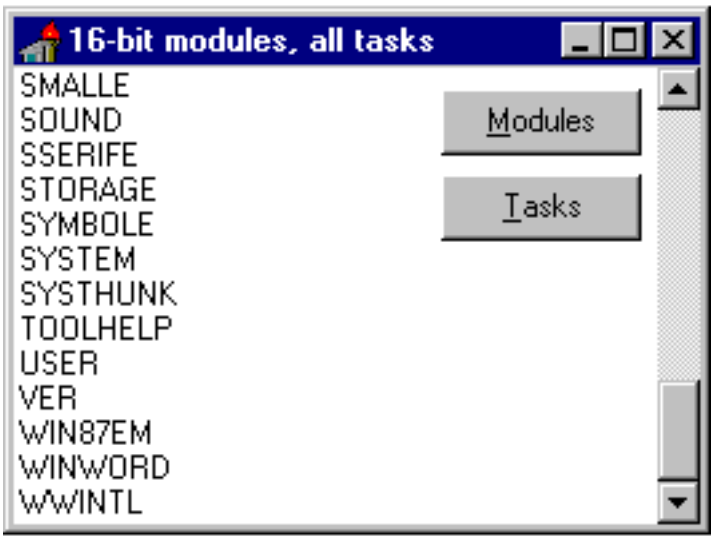
➤ *Listing 19: Housekeeping sections of a ToolHelp thunktion unit*

the 16-bit value will come through, but it checks nevertheless.

The last point is a bit of a contentious issue. The values returned by `GetFreeSystemResources` are not absolute values, as given in Windows 3.1. Instead they are relative values. When Windows 95 is launched, typically Windows Explorer is also launched as the system shell – the thing that implements the system tray and the icons on the desktop. Of course you often re-launch Explorer to act as a File Manager like application during a Windows session: it is an application with several purposes.

When Explorer has settled down, the system resources are calculated and stored away. These are used as a benchmark to compare subsequent resource values

➤ *Figure 2*




```
var ProcAddress: Pointer;
...
function ModuleFindHandle(lpModule: PModuleEntry; hModule: THandle): THandle;
var EatStackSpace: String[$3C];
begin
  // Ensure buffer isn't optimised away
  EatStackSpace := '';
  ProcAddress := GetProcAddress16(ToolHelp16Handle, 'ModuleFindHandle');
  if Assigned(ProcAddress) then begin
    MEPtr32Bit^ := lpModule^;
    asm
      push  MEPtr16Bit
      push  hModule
      mov   edx, ProcAddress
      call  QT_Thunk
      mov   Result, ax
    end;
    lpModule^ := MEPtr32Bit^;
  end;
end;
```

➤ *Listing 20: A ToolHelp thunktion*

against. This means that a value of 80% GDI resources free (as in the figure above) actually means the amount of GDI resources free are 80% of the value that were free after Explorer loaded. This can be taken to be making the figure look rather favourable, and usually rather better that we were used to in Windows 3.1, cooking the books as it were. It can also be interpreted as taking account of resources taken by Explorer that could never be recovered by the user, and which therefore have no real relevance in appearing in reported statistics.

Missing ToolHelp Functionality

The above examples were mainly for demonstration and didn't really give a good impression of how things work with real 16-bit DLLs in real applications. In this section we'll check out some of the `ToolHelp` functionality that has disappeared in the 32-bit version and see how easily we can get it back. To do this, the desirable missing 16-bit import declarations, along with their supporting types, have been copied from the original 16-bit import unit to a 32-bit thunktion unit called TLHELP16.PAS, to distinguish it from the original file TOOLHELP.PAS.

In order to get the data types used compiling down to the same memory layout in the two platforms, several steps are required.

1. The `THandle` type (which is 32-bit in Delphi 2 and 16-bit in Delphi 1) is redefined to be the same as a `THandle16` (as defined in the QTTHUNKU.PAS unit).

2. The type `Bool` used in many Windows import declarations, which is a synonym for `LongBool` in Delphi 2, but `WordBool` in Delphi 1, is redefined to be a `WordBool`.

3. Optimal 32-bit record alignment is turned off, not with the packed record modifier, as is common in Delphi 2, but with a compiler directive.

Additionally, `EBP` stack frames are ensured for the thunktions that are defined in the unit by using a compiler directive as discussed before.

Because a lot of the often called `ToolHelp` routines take pointer parameters, which are record addresses, the initialisation section of the unit allocates 16-bit accessible buffers for the various records and 32-bit equivalent pointers are also set up. Additionally, it loads up the 16-bit TOOLHELP.DLL module (which isn't present all the time, unlike the core 16-bit system DLLs User, GDI and Kernel). The finalisation section tidies all this up. Listing 19 shows what's been discussed so far.

The rest of the unit implements the thunktion wrappers. One of them is shown in Listing 20 as a demonstration of what they all look roughly like.

An example project, LISTER.DPR is written for Delphi 1 where it uses the `ToolHelp` import unit. This program lists all 16-bit modules (using `ModuleFirst` and `ModuleNext`) and all 16-bit and 32-bit tasks (using `TaskFirst` and `TaskNext`) as dictated by a couple of buttons on the form. The code that does the module/task listing is executed via a timer: as programs are loaded and unloaded the list updates itself. This ability to see 16-bit modules is not normally available to 32-bit applications. However, with a minor piece of conditional compilation, it is ready for Delphi 2 by using the `TlHelp16` thunktion unit instead (see Figure 2).

For comparison purposes, two additional projects are also supplied which use the 32-bit `ToolHelp` functionality available in Windows 95 to list 32-bit modules (using `CreateToolhelp32Snapshot`, `Module32First` and `Module32Next`) and all 16-bit and 32-bit tasks, or processes (using `CreateToolhelp32Snapshot`, `Process32First` and `Process32Next`). LISTER2.DPR is written in a similar way to LISTER.DPR, in that the relevant information is written directly into the listbox. LISTER2B.DPR takes a

```
//Alternative wrapper routine for GetFreeSystemResources
//based on how the real one works
function GetFreeSystemResources(SysResource: Word): Word;
var SHI: TSysHeapInfo;
begin
  SHI.dwSize := SizeOf(SHI);
  SystemHeapInfo(@SHI);
  //Set up a possible return value
  Result := SHI.wGDIFreePercent;
  case SysResource of
    gfsr_GDIResources: { GDI value already set up };
    gfsr_UserResources: Result := SHI.wUserFreePercent;
  else
    //If neither GDI nor User value requested
    //return the lower of the two
    if Result > SHI.wUserFreePercent then
      Result := SHI.wUserFreePercent;
  end;
end;
```

➤ *Listing 21: A possible replacement for Listing 18*

slightly different approach. To get a slight efficiency gain, it writes the information into a `TStringList` object, which is then copied into the listbox.

The efficiency gain arises because the Windows 95 listbox is implemented in the 16-bit Windows system files. Every 32-bit listbox operation is itself thunked down to 16-bit by Windows. Doing this many times in succession can get rather slow. Doing it all in one hit with a `TStringList` can speed things up.

Final Note

This new `ToolHelp` thunktion unit gives us another possible implementation of `GetFreeSystemResources`, given that the real Win16 one is implemented as a wrapper around `SystemHeapInfo`, which is catered for in `TlHelp16`. Listing 21 is an alternative definition for it, which also appears in the QTFUNCSU.PAS unit file, used by the FSR.DPR project described in a previous section (it is wrapped up in a conditional compilation section to ensure only one of the two versions is used).

References

- Matt Pietrek, *Windows 95 System Programming Secrets*, IDG Books, 1995. ISBN 1-56884-318-6.
- Matt Pietrek. *Direct Thunking in Windows 95*, Dr Dobbs Sourcebook, Volume 21, Issue 14, Number 256, March/April 1996.
- Andrew Schulman. *Unauthorized Windows 95*, IDG Books, 1994. ISBN 1-56884-169-8.

Brian Long is a freelance Delphi consultant and trainer based in the UK. He is available for bookings and can be contacted by email on 76004.3437@compuserve.com