Subclassing Windows From EXE Files In 16-Bit Applications

by John Chaytor

If you want to subclass a window from an external application in Delphi you normally need to perform this subclassing in a DLL due to the restriction imposed on the use of smart callbacks throughout the VCL. This article demonstrates a technique which does not require a DLL. You can subclass totally within an EXE and circumvent this restriction by the use of a short assembler routine.

Smart Callbacks Directive {$K}

This directive affects the way that entry code for exported functions in an EXE file is generated. It does not affect DLLs, this is why you can subclass within a DLL but not an EXE file. This directive has global scope and is implemented during the link phase. This means that a whole EXE file will either use smart callbacks or not use them: you cannot switch it on and off. When the `$K` directive is used an application does not need to call `MakeProcInstance` and `FreeProcInstance` for callback functions (although it does no harm if you do so). This option can be found on the `Compiler Options` page of the `Options|Project` dialog box.

Why Use Smart Callbacks?

Smart callbacks were implemented to avoid the need for an application to call `MakeProcInstance` for each exported callback function. Microsoft created the `MakeProcInstance` API to ensure that the DS register was set up correctly for your callback routine. However, for smart callbacks to work correctly the assumption is that the DS and SS registers have the same value. If this is not the case, it's GPF time!

How MakeProcInstance Works

To use `MakeProcInstance` you pass the address of the exported callback function and the `hInstance` variable. The function is defined:

```
function MakeProcInstance(
  Prot: TFarProc; Instance:
  THandle): TFarProc;
```

The address returned by this API call should then be passed as the address of the callback function to API calls rather than the original function address. `MakeProcInstance` creates a small piece of code which essentially does the following:

```
MOV AX,hInstance
CALL YourCallbackFunction.
```

As you can see, this routine is simply loading the AX register with the `hInstance` value (ie your application's DS register value). The entry code of your exported function can use this fact to load the DS register properly so you can access variables in your own data segment.

If you refer to page 177 of the Delphi 1 *Object Pascal Language Guide* you will see that it describes in detail the assembler code generated for the entry and exit code of an exported function. Listing 1 shows an empty exported function along with the code that would be generated by the compiler.

As you can see, that simple empty function created 17 lines of assembler code. This assumes that stack checking is disabled (more on this later). As a side point, this is why, when using the integrated debugger, if the line of code to be executed is on the `begin` statement you cannot see the contents of the local variables; when you step over the `begin` line, the local variables are available.

If you look carefully at the entry code you will see that the first line (`MOV AX,DS`) seems to be incompatible with `MakeProcInstance` as it overwrites the AX register with the DS register, which would make the code generated by `MakeProcInstance` useless! Also, line 7 of the code (`MOV DS,AX`) simply copies it back, which at face value seems

➤ *Listing 1*

```
function MyExportedFunction(X,Y: Integer): Boolean;
var I: Integer { simply to show code to allocate/deallocate locals }
begin
  { This function does nothing! }
end;

The begin statement generates the following code:

MOV AX,DS       ; Load DS selector into AX
NOP             ; Additional space for patching
INC BP          ; Indicate a far frame
PUSH BP         ; Save odd BP
MOV BP,SP       ; Set up stack frame
PUSH DS         ; Save DS
MOV DS,AX       ; Initialise DS
SUB SP,2        ; Allocate space for local variables
PUSH SI         ; Save SI
PUSH DI         ; Save DI

The end statement generates the following code:

POP DI          ; Restore DI
POP SI          ; Restore SI
LEA SP,[BP-2]   ; Deallocate space for local variables
POP DS          ; Restore DS
POP BP          ; Restore odd BP
DEC BP          ; Adjust BP
RETF 4          ; Remove parameters and return
```

like a waste of time. However, like most things in life, all is not what it seems! When Windows loads a program into memory it does one of the following:

- For exported functions in EXE files the `MOV AX,DS` at the entry point is replaced by two `NOP` instructions. This means that the `MakeProcInstance` will now work as designed: the AX register will not be overwritten.
- For exported functions in DLL files the `MOV AX,DS` and the following `NOP` instruction are replaced with `MOV AX,xxxx` where `xxxx` is the segment address of the DLL's automatic data segment (this is why the `NOP` instruction is required: `MOV AX,xxxx` requires 3 bytes whereas `MOV AX,DS` only uses two bytes).

Therefore, it's the Windows load program which makes your code compatible with `MakeProcInstance` (for DLLs `MakeProcInstance` is a waste of time anyway, it simply returns the address of the original exported function back to you).

How Smart Callbacks Work

If you have smart callbacks enabled in your Delphi installation (this is the default setting), the linker will search for all occurrences of exported functions which start with `MOV AX,DS` followed by an `NOP` instruction (for EXE files) and change this to `MOV AX,SS` plus `NOP`. The effect of this is that when Windows loads your executable into memory it will not change this code.

So, when smart callbacks are used, the AX register is set to the SS value. In line 7 of the entry code in Listing 1 the AX is copied to the DS register, the end result being that the stack and data segment registers are the same.

This is why smart callbacks make the assumption that the DS and SS registers should hold the same value. Due to the way the entry code is created you can see that `MakeProcInstance` will have no effect. The AX register which it so kindly set up for you is immediately trashed by the first line in the entry code.

Why Smart Callbacks Stop Calls From External Applications

If you attempt to subclass a window from another application using an exported function in your EXE file you will get GPFs as soon as you attempt to access any of the variables in your own data segment. This is because, on entry to your exported function, the SS segment value will be that of the caller, not yours. So the code generated when using the smart callback compiler directive will set the DS register to the same value as the SS register from the calling application. So at this point DS does in fact equal SS, but unfortunately it's not yours! The task at hand is how to get round this problem so we can export a callback from the EXE file.

If you're a bit squeamish about bending the rules you should take note of the statement on page 178 of the *Object Pascal Language Guide*: "Unless a callback routine in an application is to be called from another application (which isn't recommended anyway) you shouldn't have to ever select the {$K-} state." Now is your chance to leave...

Overcoming The SS=DS Assumption

I faced a problem in that the standard `begin` statement for the exported function will always end up with the DS register being invalid. I had to overcome this problem in some way. As I needed to know the value of the application's DS register I make use of the `MakeProcInstance` API (that's what it's for after all) to create the prolog code described previously. My next task was to replace the exported function entry code with something which provided the same functionality as the standard entry code but was compatible with `MakeProcInstance`. Listing 2 shows my new 'exported' function.

The first thing I did was to remove the old entry code by removing the `export` keyword and replacing it with `far` and `assembler` keywords. The `far` keyword will cause the compiler to generate different entry and exit code. Listing 3 (over the page) shows the code that the compiler generates (from page 177 of the *Language Guide*).

As I don't have any local variables the `SUB SP,LocalSize` code will not be generated.

I then had to manually add code to make up for the code no longer generated by the compiler automatically (since, as far as the compiler is concerned, it is not being exported – well, it *is* going to be exported, it's just that it doesn't know that!). The first 4 lines of `AsmWndProc` does this. The second line of code (`MOV DS,AX`) loads the data segment with the correct value as set up by the `MakeProcInstance` generated code. Skipping to the end of the function, I had the same problem with the exit code. I

➤ Listing 2

```
function AsmWndProc(Handle: hWnd; Msg: Word; wParam: word;
  lParam: LongInt): LongInt; far; assembler;
asm
  { Entry code to complete functionality of exported function }
  PUSH DS                  { Save caller's data segment }
  MOV  DS,AX               { Load local data, from MakeProcInstance }
  PUSH SI                  { Save caller's source index }
  PUSH DI                  { Save caller's destination index }
  { Code to call the method }
  PUSH Handle
  PUSH Msg
  PUSH wParam
  PUSH lParam.Word[2]
  PUSH lParam.Word[0]
  PUSH MyWndProcAddr.Word[6]     { Puts the Self 'hidden' parameter }
  PUSH MyWndProcAddr.Word[4]     { on the stack }
  CALL [TFarProc(MyWndProcAddr)] { Call the method }
  { Exit code to complete standard functionality of exported function }
  POP  DI                  { Restore callers destination index }
  POP  SI                  { Restore callers source index }
  POP  DS                  { Restore callers data segment }
end;
```

had to manually add the code not generated by the compiler. The last three lines of AsmWndProc do that.

To call the method I needed to push the parameters on the stack along with the object instance address (the Self parameter which all methods accept as a hidden parameter) then call the method. To do this I needed both the method address and the object address. This was done by defining a method pointer called MyWndProcAddr which is of type TMyWndProc. The interesting thing about method pointers is that they are really two pointers. The first pointer is the address of the method and the second pointer is the address of the object (ie the value of Self). You can see from Listing 3 how the Self pointer is pushed on the stack:

```
PUSH MyWndProcAddr.Word[6]
PUSH MyWndProcAddr.Word[4]
```

I then needed to call the method. I had a bit of trouble getting the compiler to accept the syntax of the CALL statement until I realised that all I needed was a typecast, as the first pointer is the one I want anyway. So the call is:

```
CALL [TFarProc(MyWndProcAddr)]
```

Stack Checking

A final problem I had in testing was that if I had the stack checking option on I got GPFs again. As I guessed, once I looked into it I found that the compiler generated extra steps in the entry code which caused the AX register to be trashed again. To get round this, I have explicitly switched the stack checking directive off just before the function. This will override any global options.

Demo Application

A demo project, SUBCLEXE.DPR is included on this month's disk which allows you to subclass windows from other 16-bit applications. The application does two things. When it receives a message it lists the details in the listbox at the bottom of the form (unless the message is one of wm_mousemove, wm_SetCursor and wm_NCHitText as there are too many of them). It keeps the last 100 messages. If the message is a wm_getminmaxinfo it will process the message and restrict the window size to the values you specify.

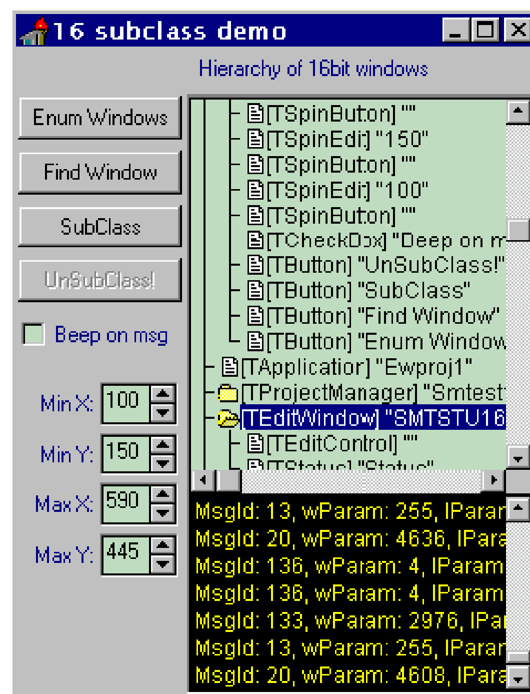
Figure 1 shows the single window displayed when you run the application.

When the form is first displayed both the outline and listbox will be empty. Click the Enum Windows button to take a snapshot of the windows currently present in the system. After a few seconds the windows will appear in a standard hierarchy of parent-child relationships in the outline window. The window class is in square brackets and the window text is in quotes. If you are running on Windows95 I attempt to identify if the window is a 16-bit window so that 32-bit windows do not appear in the list. See the IsWindow16Bit function in the unit. I don't know if it's 100% but then again, this is just a demo!

You can choose one of the windows to subclass by selecting it in the outline window then clicking on the Subclass button. To make it easier to identify windows I have added the FindWindow button. When you click this button the cursor will change to a cross and the mouse movements will be captured. As you move it over windows the relevant window will be selected in the outline window. Click the left mouse button when you are over the window of interest then click the Subclass button to start getting that window's messages.

To process the wm_getminmaxinfo messages you really need to get a parent form (whose parent is the desktop) which is sizeable (an

➤ *Figure 1*



➤ *Listing 3*

```
The asm statement generates the following code:

  INC BP       ; Indicate a far frame
  PUSH BP      ; Save odd BP
  MOV BP,SP    ; Set up stack frame
  PUSH DS      ; Save DS

The end statement generates the following code:

  MOV SP,BP    ; Remove locals and saved DS
  POP BP       ; Restore odd BP
  DEC BP       ; Adjust BP
  RETF 10      ; Remove parameters and return
```

example is Delphi's own edit window of class TEditWindow). When you subclass the window and attempt to resize it, it will be restricted to the values in the spin edit controls.

Initialising, Subclassing And Tidying Up

As MakeProcInstance should only ever be called once for the life of an application I do this in the FormCreate method. This variable is defined as type TFarProc.

You can see from the code on the disk that the enumeration of the 16-bit windows is done in three methods (BtnEnumClick, EnumWindowsProc and EnumChildProc). Interestingly, the last two are callback functions called by windows which make use of the smart callback option I'm trying to circumvent elsewhere in the program. I don't need to call MakeProcInstance for these. To identify 16-bit windows I call SetWindowLong for each window (passing the original WndProc address so no damage is done) because the value returned is nil if you attempt this for a 32-bit window. I don't know if there is a better way to do this.

When subclassing we simply do the standard SetWindowLong and pass the address which is returned from MakeProcInstance. The old window's procedure is stored.

For 'un-subclassing' (I'm not too sure what the official term is for this but 'un-subclassing' seems OK to me!) we just call SetWindowLong and pass the original value to unhook us – Ahah, perhaps that's the term!

FreeProcInstance is called during FormDestroy to clean up the resource.

That's about it really. Just a quick and dirty method of subclassing windows without the need to use a DLL.

John Chaytor lives in Brighton, England, and can be contacted via CompuServe as 100265,3642