Generic MakeMethodInstance For 16/32 Bit Applications

by John Chaytor

Aren't you just sick of the way that, whenever you want to use that handy little routine that someone else has written, the job becomes ever more complicated as they rely on callback routines which, unfortunately, cannot accept an object method as a parameter? Well I am! The routines were probably written well before Delphi made an appearance on the scene, so they know nothing of the calling conventions required for Delphi methods.

I faced such a problem when, after creating various classes, I needed to use some of the vendor's utility tools to maintain my databases. Unfortunately, all the routines I needed to use relied heavily on callback functions. Also, the routines I would like to make use of in the callback functions were hidden away well within objects which made it difficult to get at them from outside the class. After a few choice words I decided to do a bit of investigation.

What I needed was the ability to call these routines from within the class, passing a method for the callback. I knew of the built-in `MakeObjectInstance` function, which allows you to have a method indirectly called from Windows, so I knew what I had in mind could be done. It quickly became apparent that the problem would be easier to solve than I had thought.

So What Is The Problem?

The problem arises because Delphi methods accept one more parameter than a standard Pascal function.

Consider the following declaration for a function which returns the maximum integer of the two passed:

```
function MaxInt(
  A, B : Integer) : Integer;
```

This function expects two integer parameters to be passed on the stack. Now, if the same function were a method of a class (Ok, so it's a silly example!):

```
function TMyClass.MaxInt(
  A, B : Integer) : Integer;
```

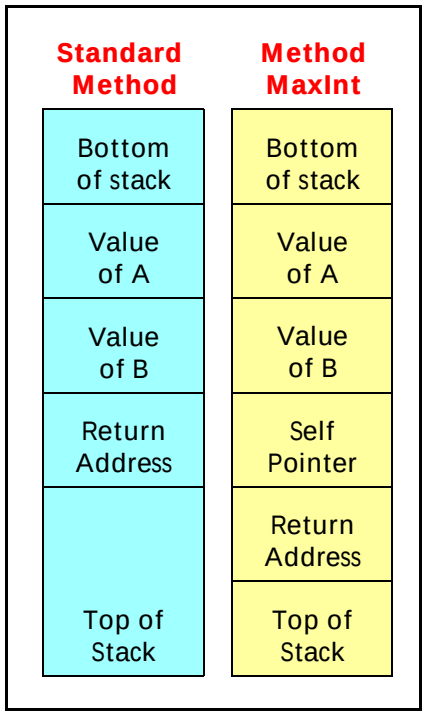
it would expect *three* parameters on the stack: the two integers as before, plus a hidden parameter which identifies the instance of the object. This is the `Self` variable which is available in all methods even though you never see it explicitly declared anywhere.

So, because a method expects an additional parameter the above two functions are not equivalent and the compiler will stop you from using one when the other type is required. Even if you managed to fool the compiler by typecasting or some other underhand way you are going to face grief sooner or later as the stack will not be being processed correctly. Note that you don't need to worry about the result as it is passed back in a CPU register, not on the stack.

The Solution

What we really need to do is insert the `Self` parameter into the stack whenever our function is called from outside the class (yes, you read that correctly, 'insert' is what I said!). This will allow all methods to work correctly as they will have the correct value for the `Self` variable. Going back to our two examples for the `MaxInt` function, Figure 1 shows the state of the stack at the beginning of each function before any code is executed.

As you can see, the `Self` parameter for the `MaxInt` method has been placed at the end of the explicitly declared parameters but before the return address (the stack grows downwards, this is why *Top of stack* appears below *Bottom of stack* in the figure). This illustrates graphically why the method and standard functions are incompatible. It is the `Self` parameter which will be inserted between the last parameter (*B* in Figure 1) and the return address, by manipulating the stack.



➤ *Figure 1*

General Overview

This section gives a general conceptual overview of how the problem was tackled, as there are differences in implementation between Delphi 1 and Delphi 2. The general concept is that the routine dynamically builds a small machine code routine which changes the contents of the stack then calls our original method. It is not restricted by the number and types of parameters passed.

For example, if this routine were implemented for the `MaxInt` method in Figure 1 the stack will be changed from the Standard to the

Method version in this stub routine prior to executing the method. So the method never knows we were involved.

We will create a new function called MakeMethodInstance which will do the following...

Firstly, we allocate just a few bytes of memory to contain the dynamically created stub code. We only need 13 bytes for Delphi 2 and 19 bytes for Delphi 1. This memory must remain at a fixed address until the callback is no longer active.

Then we ensure that the memory is allowed to be executed. When you allocate memory it is normally used for data. Due to this, Windows will stop this memory from being executed as a program. If you attempt this via a call or jump instruction you will get a GPF. However, this is exactly what we need to do so we use the facilities of the operating system to ensure that the memory we have just allocated is marked as executable. When it is later called Windows will be happy that the memory has execute privilege and won't complain.

Next we must dynamically build machine code instructions and place them in memory. This is what the code is trying to achieve:

- Remove the return address off the stack and store it.
- Put the object instance (Self) variable onto the stack.
- Put the return address back on the stack.
- Jump to the original method entry point.

Luckily, due the fact that the Self parameter is after the explicitly declared parameters, the routine is not dependant upon the number or types of parameters. It will work for any method.

To make this easier to implement and understand I have created record structures for both implementations then built the machine code by populating the record structure with the values which represent the machine code instructions and parameters. I assure you that this will be easy to understand once you see the code!

Finally we return the address back to the caller. After we have allocated the memory, ensured that it is executable and built the required machine code, we pass the address of this memory back to the caller. This address should then be passed to the callback routine. When this is called from the external program it will execute the machine code we have just created. This is why the memory must remain at a fixed position as long as the callback is active.

When the callback is no longer required we should call FreeMethodInstance to free up all the resources which were allocated by MakeMethodInstance.

Listing 1

```
type
  PJumpBlock = ^TJumpBlock;
  TJumpBlock = packed record
    POP_EAX_OpCode: Byte;
    Push_Immed_OpCode: Byte;
    Self_Value: Pointer;
    PUSH_EAX_OpCode: Byte;
    Jmp_OpCode: Byte;
    Method_Addr: Pointer;
    DummyAddr: Byte;
  end;
function MakeMethodInstance(Code,Data: Pointer): Pointer;
begin
  Result := VirtualAlloc(nil, sizeof(TJumpBlock),
    MEM_COMMIT, PAGE_EXECUTE_READWRITE);
  if Result <> nil then
    with PJumpBlock(Result)^ do begin
      POP_EAX_OpCode     := $58;  { POP Return address into EAX register }
      Push_Immed_OpCode := $68;  { PUSH DWORD following this instruction }
      Self_Value         := Data; { Set DWORD to object instance address }
      PUSH_EAX_OpCode    := $50;  { Push the return address back on stack }
      Jmp_OpCode         := $E9;  { JMP to relative offset following this opcode }
      Method_Addr        := Pointer(LongInt(Code)-LongInt(@DummyAddr));
    end;
end;
procedure FreeMethodInstance(Instance: Pointer);
begin
  if Instance <> nil then
    VirtualFree(Instance,0,MEM_DECOMMIT);
end;
```

32-Bit Implementation

Listing 1 shows the code specific to the 32-bit implementation. You can see the first thing this method does is allocate memory using the VirtualAlloc API. Refer to the on-line help for additional information on this call. All we need to concern ourselves with here is that the memory allocated is of type PAGE_EXECUTE_READWRITE. This means it can contain executable code. We then populate the record structure with valid Intel machine code. All the fields with the suffix OpCode will be set to values which are machine code instructions, the others are the parameters those instructions use (some OpCodes don't have any parameters). The comments indicate what each step does. Remember that, although you are looking at a record structure, this will be executed by the CPU.

If you don't know (or don't want to know!) assembler don't be too concerned: all you need to know is that the stack holds temporary data (the last value added will be the first removed), POP removes data off the stack, PUSH puts data on the stack and JMP jumps to the specified address. From the comments you should get a rough idea of what is being done.

We will step through the 32-bit version in detail to see how it builds the machine code and how each step affects the stack. The 16-bit version follows exactly the same principle so will not be described in such detail. This discussion assumes we are referring to the MaxInt function described earlier (see Figure 1).

Building The Machine Code

Each field of the record structure TJumpBlock is populated in turn to create the machine code. The field POP_EAX_OpCode is set to $58. When the CPU executes this instruction it

will remove the DWORD off the top of the stack (which is the caller's return address) and put it in the EAX register. The return address is no longer on the stack after this instruction: B is now at the top of the stack. The next field to be populated is Push_Immed_OpCode, which is set to $68. When the CPU executes this instruction it will take the DWORD immediately following the instruction and place it on the stack. As the field following Push_Immed_OpCode is Self_Value we need to set that to the object instance address. This is why we pass Self to the MakeMethodInstance function (as the Data parameter). The value passed is copied to the field Self_Value. Therefore, when the CPU executes the $68 opcode Self is placed on the stack. Now, at this point Self is at the top of the stack. All that remains to be done, as far as the stack is concerned, is to put the return address back on the stack.

As the CPU knows that the $68 instruction is followed by a DWORD parameter, the next instruction it executes will be the contents of PUSH_EAX_OpCode. This is set to $50. When the CPU encounters this instruction it pushes the content of the EAX register onto the stack. As we had previously popped the return address into this register, it is placed back on the stack. At this point, the stack has been converted from the standard function style to the method style (see Figure 1). Now, the only problem remaining is that we need to get the CPU to start executing the code for the original method. The field Jmp_OpCode is the next instruction that will be executed and is set to $E9. This instruction causes the CPU to start executing code at a different address. To calculate the value of that address it will take the DWORD immediately following this instruction and start executing code at that relative address (forwards or backwards in memory). This is where the DummyAddr field is used, simply as a means of calculating the relative offset of the method entry point (the code parameter) to the current CPU execution address. As you can see from the code in Listing 1, some simple arithmetic is used to calculate the relative offset and this is assigned to the Method_Addr field. We have now completed the creation of the machine code stub. The memory now contains a fully functional piece of machine code which can be called by callback functions.

When the callback is no longer required FreeMethodInstance is called which simply frees the memory, calling VirtualFree.

Optimisation

Delphi 2 performs various optimisation tricks, one of which is to pass up to three parameters in registers rather than on the stack. The parameters are placed in the EAX, EDX and ECX registers (in that order) as required. For a method the relative position of the Self parameter is dependent upon the calling convention used.

If a method is declared with the register keyword (it is by default if you don't specify pascal, cdecl or stdcall) then the Self parameter is optimised and processed before the explicitly declared parameters and is passed in the EAX register. Up to two of the remaining explicitly declared parameters can be

➤ *Listing 2*

```
type
  PJumpBlock = ^TJumpBlock;
  TJumpBlock = packed record
    POP_AX_OpCode: Byte;
    POP_CX_OpCode: Byte;
    Push_Seg_Immed_OpCode: Byte;
    Self_Seg_Value: Word;
    Push_Ofs_Immed_OpCode: Byte;
    Self_Ofs_Value: Word;
    PUSH_CX_OpCode: Byte;
    PUSH_AX_OpCode: Byte;
    Jmp_OpCode: Byte;
    Method_Addr: Pointer;
    { Specific information needed for 16 bit segmented memory }
    DataSelector: THandle;
    CodeSelector: THandle;
  end;
function MakeMethodInstance(Code,Data: Pointer): Pointer;
var
  WrkHData,WrkHCode: THandle;
begin
  WrkHData := GlobalAlloc(HeapAllocFlags,SizeOf(TJumpBlock));
  Result := GlobalLock(WrkHData);
  if Result <> nil then
    With PJumpBlock(Result)^ do begin
      POP_AX_OpCode := $58;    { POP Return address ofs into EAX register }
      POP_CX_OpCode := $59;    { POP Return address seg into ECX register }
      Push_Seg_Immed_OpCode := $68;  { PUSH Self segment value onto stack }
      Self_Seg_Value        := PtrRec(Data).Seg;
      Push_Ofs_Immed_OpCode := $68;  { PUSH Self segment offset onto stack }
      Self_Ofs_Value := PtrRec(Data).Ofs;
      PUSH_CX_OpCode := $51;    { PUSH the CX register back onto the stack }
      PUSH_AX_OpCode := $50;    { PUSH the AX register back onto the stack }
      Jmp_OpCode     := $EA;    { JMP to the address following this opcode }
      Method_Addr    := Code;
      WrkHCode       := AllocDsToCSAlias(PtrRec(Result).Seg);
      PtrRec(Result).Seg    := WrkHCode;
      { Store the code and data selectors for FreeMethodInstance }
      DataSelector   := WrkHData;
      CodeSelector   := WrkHCode;
    end;
end;
procedure FreeMethodInstance(Instance: Pointer);
var
  WrkHData,WrkHCode: THandle;
begin
  if Instance <> nil then
    With PJumpBlock(Instance)^ do begin
      WrkHData := DataSelector;
      WrkHCode := CodeSelector;
      GlobalUnlock(WrkHData);
      GlobalFree(WrkHData);
      FreeSelector(WrkHCode);
    end;
end;
```

Number of parameters optimised	Processing involved
0	Copy Self value to EAX JMP to original entry point
1	Copy EAX register to EDX Copy Self value to EAX JMP to original entry point
2	Copy EDX register to ECX Copy EAX register to EDX Copy Self value to EAX JMP to original entry point
3	'Insert' ECX register into stack Copy EDX register to ECX Copy EAX register to EDX Copy Self value to EAX JMP to original entry point

➤ *Table 1*

optimised into the EDX and ECX registers. This is incompatible with the way MakeMethodInstance works. Therefore, for MakeMethodInstance to work, both the callback type definition and the callback method must specify the pascal keyword to ensure that the parameters are passed on the stack (see later for details of an enhanced routine which does cater for the register calling convention).

16-Bit Implementation

Listing 2 shows the code specific to the 16-bit implementation. You will see that there are several changes, which are all related to the specifics of 16-bit memory management. To allocate the memory in the 16-bit version we call GlobalAlloc and GlobalLock. This results in a piece of memory which is in a fixed place but without execute privilege.

To obtain execute privilege for the memory we call AllocDsToCSAlias which accepts a selector for a data segment and returns a different selector to the same memory: the difference being that the new selector has execute privilege for that memory. The segment value of the result is then changed to this 'execute enabled' selector rather than the original data selector returned from the GlobalLock function. This ensures that the callback address used has execute privilege (if the original data selector were used a GPF would be generated even though the same block of memory was being executed).

Two extra fields have been added to the TJumpBlock record structure (DataSelector and CodeSelector) to store the selector handles which are required in FreeMethodInstance.

If you look at the way the machine code is built up, each PUSH and POP has been done in two steps as the addresses are manipulated in WORD rather than DWORD values. Also the jump instruction has been changed to use the actual address rather than a relative address. However, the functionality of this code is exactly the same as the 32-bit version.

De-allocating memory in the 16-bit version of FreeMethodInstance takes more steps due to the need to free selector handles. FreeMethodInstance first takes a copy of the two selector values from the dynamically allocated memory then unlocks that memory (which is now invalid: that is why we took a copy of the selectors). It then frees the two selectors to ensure Windows resources are released.

Enhancements

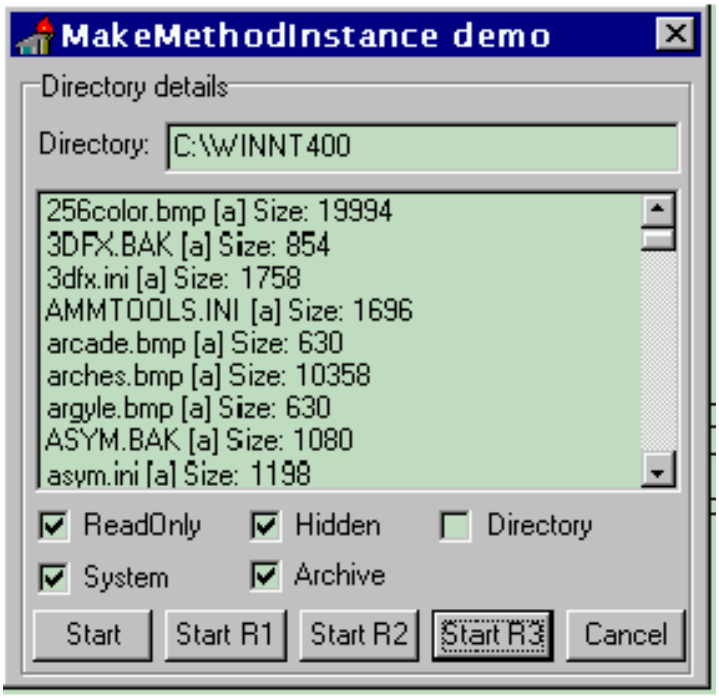
For the 16-bit implementation, MakeMethodInstance uses two selectors each time it is called, as it allocates a new block of memory. An enhancement would be to allocate a block of memory on the first call and sub-allocate parts of it for each new caller. This would reduce the number of selectors used. Take a look at the source for MakeObjectInstance to see how this has been done in that function. I didn't add it to this version of MakeMethodInstance as it would complicate the discussion. This same approach could be taken for the 32-bit version as VirtualAlloc will always allocate at least 4Kb of memory. This has been left as an exercise for the reader...

I have created an enhanced version of the routine called MakeMethodInstance32Reg which caters for the register calling convention used in Delphi 2. This could be used if you do not want to change the source code (or cannot, if, for example, the unit has been precompiled) to use the pascal calling convention. This function accepts an additional parameter which is the number of parameters optimised into the registers. It then changes the register values and stack as required. Unfortunately, there is no way to programmatically determine how many parameters have been optimised. You should refer to Chapter 17 of the *Language Guide* for guidance. This value must be between 0 and 3.

Table 1 details the processing performed in the stub code created by MakeMethodInstance32Reg depending upon the number of optimised parameters. The basic processing is that the Self variable is placed in the EAX register after the other optimised parameters have been shifted to their new position (ie EAX is copied to EDX, EDX is copied to ECX and ECX is inserted in the stack as required). Note that, unless three parameters have been optimised, the stack is not changed in any way. The stub code simply changes register values then jumps to the original method entry point. If three parameters have been optimised then the stack does needs to be changed.

Demo Application

A demonstration project is included on this month's disk, called MAKEMI.DPR, which can be compiled in Delphi 1 or 2. It consists of four files. MAKEMIU.PAS/DFM is the main form, MAKEMIC.PAS contains the MakeMethodInstance, MakeMethodInstance32Reg and FreeMethodInstance functions and the



➤ *Figure 2*

```
{ From DIRLIST.PAS }
type
  { TScanDirCallBack is defined with the pascal calling convention for
    Win32. The 16 bit version does not specify this keyword as it is
    invalid in that environment and all parameters are passed on the stack }
{$IFNDEF WIN32}
  TScanDirCallBack = function(CurrentFile: string; Attr: Byte): Boolean;
{$ELSE}
  TScanDirCallBack =
    function(CurrentFile: string; Attr: Byte): Boolean; pascal;


{ From MAKEMIU.PAS }
{ FCallBack is declared as: }
FCallBack: TScanDirCallBack;
{ The Start button event procedure contains: }
MakeMethodInstance(@TForm1.ScanDirCallBack,Self);
ScanDir(BaseDirectory.Text,FCallBack);
FreeMethodInstance(@FCallBack);
{ The callback method is declared as follows: }
{$IFNDEF WIN32}
 function TForm1.ScanDirCallBack(CurrentFile: string; Attr: Byte): Boolean;
{$ELSE}
 function TForm1.ScanDirCallBack(CurrentFile: string; Attr: Byte):
   Boolean; pascal;
{$ENDIF}
```

➤ *Listing 3*

file DIRLIST.PAS contains the functions which call the callback routines. Figure 2 shows the form displayed in the 32-bit version (the 16-bit version will not have the `Start R1`, `Start R2` and `Start R3` buttons).

When you click the `Start` buttons a function called `ScanDir` is called which simply passes all files in the specified directory to the supplied callback function in turn. The callback function can return `True` if it wishes the `ScanDir` function to stop processing. Listing 3 shows the important pieces of code.

Referring to Listing 3 you can see that the only difference in the `TScanDirCallBack` procedure type declaration between Win32 and Win16 is that the former has the `pascal` keyword. This ensures that both pass all parameters on the stack which, you may recall, is a requirement of `MakeMethodInstance`. You should also note that the actual declaration for the callback method is defined in the same way.

In the event procedure, `MakeMethodInstance` is called passing the method address and the object instance: this is stored in `FCallBack`. Then `ScanDir` is called, passing the specified directory and the address of the stub code created by `MakeMethodInstance`. When `ScanDir` processes each file in the directory `ScanDirCallBack` will be called for each: this function can return `True` to stop `ScanDir` from processing any more files (an artificial delay has been built into the callback method to give you a chance to do this before `ScanDir` ends). When `ScanDir` ends the code in the event procedure calls `FreeMethodInstance` to free Windows resources.

In Delphi 2 the `StartR1, ...2, ...3` buttons perform the same function as the `Start` button. The only difference is that the callback functions have been changed for each one to ensure that parameters are optimised into registers (one for `Start R1`, two for `Start R2` and three for `Start R3`). In these cases `MakeMethodInstance32Reg` is called instead of `MakeMethodInstance`. By examining the code you can see that the procedure declaration for each has been changed accordingly.

Conclusions

I hope that you have found this article both informative and useful. You should certainly be able to use the functions provided (even if you don't understand a word of how it is done!) by referring to the demonstration application provided.

John Chaytor is a freelance programmer who lives and works in Brighton, UK, and can be contacted via CompuServe as 100265,3642