DelayLoading Of DLLs

Curing the VC++6.0 envy

by Hallvard Vassbotn

I don't miss many features from Microsoft's Visual C++ 6.0 when working in Delphi, but the new `/DELAYLOAD` option of the linker is one of them. It lets you turn normal, implicit DLL import libraries into so-called delayload import libraries. This means that the DLL will not be loaded by the operating system during startup of the EXE file, but rather as needed, when you actually call the routines. The first time a specific DLL routine is called, the DLL is loaded with `LoadLibrary` and the routine address is retrieved with `GetProcAddress`. This is accomplished simply by turning on the `/DELAYLOAD` option of the linker, specifying what DLLs you want to be delayloaded.

In this article, we will show how we can implement a framework for obtaining similar behaviour in our Delphi applications.

References

Both Jeffrey Richter and Matt Pietrek document the new `/DELAYLOAD` feature in the December 1998 issue of MSJ[1]. As usual, Pietrek gives the most detailed technical description, right down to the assembly code thunks created by the linker. In Issue 7 of *Developers Review* (www.itecuk.com), Dave Jewell also wrote favourably about this feature in his review of VC++ 6.0 under the sub-heading *A Linker To Die For*.

In the infancy of *The Delphi Magazine*, back in Issue 1 (April 1995), Dave Jewell started up his *Delphi Internals* column with the article *Using and Writing DLLs*. And in the following issue Bob Swart demonstrated how to explicitly load DLLs at runtime to allow for more flexible applications in his *Tic-Tac-Toe* article. We will not duplicate all of the information in those articles, but let us quickly go through the classical ways of accessing routines in a DLL.

Implicit Loading

The most common way of using a DLL is to let the operating system implicitly load the DLL when the EXE file is launched. This is achieved by declaring the routine with the `external` directive, specifying the name of the DLL:

```
procedure MyRoutine(
  A, B, C: integer);
  external 'MyDll.Dll';
```

The operating system will search for the DLL in the EXE directory, current directory and the system directories. If the DLL cannot be found, the operating system will issue an error message and refuse to launch the application. This is one of the major drawbacks of using implicit loading: you cannot have a flexible application that will work even if the DLL is not present.

By default, the compiler will use the name of the routine as given in the Pascal declaration. If the actual name in the DLL differs from this, we can use the `name` directive to override it. For instance, many of the Win32 API routines actually have names ending with an `A` (for ANSI) instead of the more well known names. The aliasing is done like this:

```
function PeekMessage;
  external 'user32.dll'
  name 'PeekMessageA';
```

Note that the names of exported DLL routines are case sensitive. Routines can also be referenced by an ordinal number. Sometimes, routines are exported using only ordinal numbers. We might also want to import routines by ordinal value to speed up the linking process or make it harder for hackers to determine the names of routines we are using. In any case, to import a routine by ordinal value, we use the `index` directive:

```
procedure MyRoutine(
  A, B, C: integer); external
  'MyDll.Dll' index 14;
```

Normally, we don't declare the DLL routines each time we want to use them, instead, we create an import unit that declares all the routines in a particular DLL. Over time there might be new versions of the DLL with new routines included. Unfortunately, if we create an import unit declaring all the routines in the latest version of the DLL, the application will not load if only an older version of the DLL is present. The problem is that if the operating system cannot find *all* the imported entries it will refuse to load the application.

Classical Explicit Loading

To overcome the shortcomings of the implicit loading technique, we can get a higher level of control by loading the DLL (using `LoadLibrary`) and linking to the routines (using `GetProcAddress`) ourselves. By doing this we can get the following potential benefits: loading the DLL only when necessary, disabling features in missing DLLs, checking the DLL version (only linking to and calling supported routines) and unloading the DLL when it is no longer needed.

There are different ways to implement explicit loading. The two main strategies differ in when to load the DLL and when to link to the routines. The simplest technique does all of the linking at initialisation time. This is the approach taken by Bob Swart in his *Tic-Tac-Toe* article, see the sample in Listing 1.

This allows the program to check the `MagicLoaded` variable and

1. *Microsoft Systems Journal*, Vol 13 No 12, December 1998.
Win32 Q&A by Jeffrey Richter (http://www.microsoft.com/msj/1298/win32/win321298top.htm)
Under The Hood by Matt Pietrek (http://www.microsoft.com/msj/1298/hood/hood1298top.htm)

simply disable the options in the application that rely on the DLL, should it be missing. This approach works fine in most situations, but if not all of the functions are available in the DLL (it could be an older version, for instance), the corresponding procedural variable will be set to `nil`. If the application calls the routine, it will fail with an Access Violation. In addition, the unit loads the DLL at startup, so the memory and loading time overhead is the same, even if the DLL is never used.

A slightly more advanced technique only loads the DLL the first time it is needed and only links in the routines that are actually called. An example of this can be seen in the import unit for MAPI, supplied with Delphi 2, 3 and 4. To import a single routine, the code in Listing 2 is required.

As you can see, there is quite a lot of code here just for importing a single routine. The rest of the unit contains almost identical code to import another 13 routines; the unit totals almost 600 lines. So even if the advantages of using explicit loading are intriguing, the overhead of implementing it like this makes you think twice. We will now see how we can simplify writing explicit import units while still keeping all the stated benefits, adding a couple more of our own.

Improved Explicit Loading

How can we keep the logic of only loading routines as they are called, but reduce the amount of code we must write for each imported routine? The trouble with the MAPI example in Listing 2 is that it always goes through the wrapper routine in the `implementation` part. What if we move the procedural variables into the `interface` section? Then we can initialise them to a corresponding thunking routine that is responsible for linking to the DLL routine the first time it is called. The thunking routine fixes up the variable and then calls through it to go to the DLL. The next time a call is made through the procedural variable, it goes directly to the DLL. See Listing 3 for an example implementation.

```
unit MAGIC;
Const
  MagicLoaded: Boolean = False;
var
  NewGame: function: HGame;
  ...
implementation
begin
  DLLHandle := LoadLibrary('MAGIC.DLL');
  if DLLHandle >= 32 then begin
    MagicLoaded := True;
    @NewGame := GetProcAddress(DLLHandle,'NEWGAME');
    ...
  end;
end.
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
interface
...
type
  PFNMapiLogon = ^TFNMapiLogOn;
  TFNMapiLogOn = function(ulUIParam: Cardinal; lpszProfileName: LPSTR;
    lpszPassword: LPSTR; flFlags: FLAGS; ulReserved: Cardinal;
    lplhSession: PLHANDLE): Cardinal stdcall;
function MapiLogOn(ulUIParam: Cardinal; lpszProfileName: LPSTR;
  lpszPassword: LPSTR; flFlags: FLAGS; ulReserved: Cardinal;
  lplhSession: PLHANDLE): Cardinal;
implementation
var LogOn: TFNMapiLogOn = nil;
function MapiLogOn(ulUIParam: Cardinal; lpszProfileName: LPSTR;
  lpszPassword: LPSTR; flFlags: FLAGS; ulReserved: Cardinal;
  lplhSession: PLHANDLE): Cardinal;
begin
  InitMapi;
  if @LogOn = nil then
    @LogOn := GetProcAddress(MAPIModule, 'MAPILogOn');
  if @LogOn <> nil then
    Result := LogOn(ulUIParam, lpszProfileName, lpszPassword, flFlags,
      ulReserved, lplhSession)
  else Result := 1;
end;
```

```
unit ExpImpTestDll;
interface
var Routine1 : procedure (A, B, C, D: integer);
...
implementation
...
// <-- Support-code goes here
...
procedure Routine1_Thunk(A, B, C, D: integer);
begin
  Routine1 := GetTestDllFunc('Routine1');
  Routine1(A, B, C, D);
end;
...
initialization
  Routine1 := Routine1_Thunk;
  ...
end.
```

➤ *Listing 3*

This works quite neatly. The startup time is reduced to practically zero, the DLL is not loaded until one of the routines in it is called and if the DLL or routine is missing, a nice exception is raised (inside the `GetTestDLLFunc` routine, see the disk for details).

However, I'm still not satisfied. We still have to write boilerplate code for each imported routine, the name of the routine is repeated 7 times and we have 6 lines of code for each import. In addition, there is a fixed overhead of support code. Contrast this with the simplicity of the implicit import unit in Listing 4. To get a correspondingly smooth way of writing explicit import units, we must take off the silk gloves and sharpen our hacking skills...

Effortless Explicit Loading

The goal we should set is to write a support unit that will enable us to do dynamic explicit linking just as easily as implicit linking. For the solution that we will go through, we will actually achieve all the stated benefits while being able to write simple import units like the one in Listing 5.

What we have done is declare the routines as procedural variables with the correct parameter

```
unit ImpImpTestDll;
interface
procedure Routine1(A, B, C, D: integer);
...
implementation
procedure Routine1; external 'TestDll.Dll' name 'Routine1';
...
end.
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
unit DynLinkTest;
interface
uses
  HVDll;
var
  Routine1 : procedure (A, B, C, D: integer); register;
  Routine2 : procedure (A, B, C, D: integer); pascal;
  Routine3 : procedure (A, B, C, D: integer); cdecl;
  Routine4 : procedure (A, B, C, D: integer); stdcall;
  TestDll: TDll;
implementation
var
  Entries : array[1..4] of HVDll.TEntry =
    ((Proc: @@Routine1; Name: 'Routine1'),
     (Proc: @@Routine2; Name: 'Routine2'),
     (Proc: @@Routine3; ID  : 3),
     (Proc: @@Routine4; ID  : 4));
initialization
  TestDll := TDll.Create('Testdll.dll', Entries);
end.
```

and calling convention signature in the `interface` section of the unit. We also declare an instance of the `TDll` class declared in the `HVDll` unit. This is not strictly required, but it can be useful for gaining greater control.

Then in the `implementation` section we declare an initialised array of `TEntry` records. There is one slot in the array for each routine. We give the address of each routine's procedural variable and the name of the routine as exported from the DLL. We can also import the entry by ordinal by using the `ID` field of the `TEntry` record. In the `initialization` section, we create the `TDll` instance, sending in the default name of the DLL and the array of `TEntry` records.

That's it. It is really that simple. And it works!

When launching an application using this unit, Testdll.Dll will not be automatically loaded. The application can change the path to the DLL, using the `FullPath` property of the `TestDll` object. The first time a call is made through one of the procedural variables, the DLL will be loaded with `LoadLibrary` and that routine will be linked to using `GetProcAddress`. The procedural variable will have its contents patched, so the next time the call is made it goes straight to the DLL code.

The DLL can later be unloaded by calling `TestDll.Unload`. The application could even decide to use another version of the DLL in mid session by assigning to the `FullPath` property again. We will now see how this kind of delayed loading is implemented.

Let's look under the hood to see how the procedural variables are magically linked to the DLL routines as they are being called.

Let The Magic Begin

The only code that runs at initialisation time is the call to the `TDll` constructor, see Listing 6.

This code is trivial. It keeps the name of the DLL and a pointer to the array of `TEntry` records before it calculates the number of entries in the array. After calling a couple of interesting named methods, it adds itself to a global list of `TDll` instances.

Generating Code On The Fly

To allow us to use the procedural variables without more code than we saw in Listing 5, we must somehow dynamically compile code thunks similar to the ones we saw back in Listing 3. This requires acting like a mini-compiler and generating executable code on the fly. To complicate matters, we have to preserve the stack layout and the contents of parameter passing registers (`EAX`, `EDX` and `ECX`). A single set of code must handle all cases of calling conventions and parameters.

As the first step, the `CreateThunks` method is responsible for dynamically creating these code thunks. Essentially, it allocates a block of memory and then fills it with CPU instruction opcodes, see Listing 7.

First, we get a memory block large enough for the thunking code. To help us get a memory block that can be both modified and executed, we use the `CodeHeap` object maintained by the DLL's instance (see *Proper Code Generation* on page 42 for details).

Then we fill the block with the CPU instructions needed to perform the thunking. For this purpose we have defined a record structure with appropriately named fields. There is a fixed 10-bytes per DLL thunk consisting of the following assembly code:

```
@@Header
  PUSH    Self
  JMP     ThunkingTarget
```

This code is responsible for pushing the address of the `Self`-pointer (the `TDll` instance pointer) to the stack and then transferring control to the `ThunkingTarget` global procedure. This allows `ThunkingTarget` to easily know the `TDll` object this call was made for and thus what DLL it needs to link to.

After this fixed header, there is an array of 5-bytes per routine thunks, like this:

```
CALL    Header
```

➤ *Listing 6*

```
constructor TDll.Create(const DllName: string; const Entries: array of TEntry);
begin
  inherited Create;
  FFullPath := DllName;
  FEntries  := @Entries;
  FCount    := High(Entries) - Low(Entries) + 1;
  CreateThunks;
  ActivateThunks;
  Dlls.Add(Self);
end;
```

This instruction simply calls back up to the per-DLL header. The purpose of the call is to have the CPU push the return address for this instruction to the stack. We will never actually return to this address, but we will use it as a base value from which we can calculate the index of the DLL routine we are calling.

Therefore, if we are importing three routines from a DLL, the total dynamically generated code will look like this:

```
@@Header
  PUSH    Self
  JMP     ThunkingTarget
  CALL    Header
  CALL    Header
  CALL    Header
```

Now that we have some properly encoded machine instructions in memory, we are ready to assign the procedural variables their initial value. This is done in the `ActivateThunks` method, see Listing 8.

Here we simply loop through all the `TEntry` records in the `FEntries` array and patch the procedure variables to point to the generated thunks. After this, the procedural variables are ready to be used.

Inside ThunkingTarget

When one of the procedural variables is called through, control will be transferred to the corresponding thunk. From here it calls back up to the per-DLL header, pushes the `Self`-pointer and jumps on to the `ThunkingTarget` procedure. This procedure is a bit tricky and it has to be written in assembly to allow us to save the contents of certain registers, see Listing 9.

It is not obvious what this code does, so let's take it step by step.

First, we push the contents of the `EAX`, `EDX` and `ECX` registers to the stack. We have to preserve the contents of these registers, because they could contain parameter values if the DLL routine we are going to call uses the register calling convention:

```
PUSH    EAX
PUSH    EDX
PUSH    ECX
```

```
procedure TDll.CreateThunks;
const
  CallInstruction = $E8;
  PushInstruction = $68;
  JumpInstruction = $E9;
var
  i : integer;
begin
  Dlls.CodeHeap.GetMem(FThunkingCode,
    SizeOf(TThunkHeader) + SizeOf(TThunk) * Count);
  with FThunkingCode^, ThunkHeader do begin
    PUSH   := PushInstruction;
    VALUE  := Self;
    JMP    := JumpInstruction;
    OFFSET := PChar(@ThunkingTarget) - PChar(@Thunks[0]);
    for i := 0 to Count-1 do
      with Thunks[i] do begin
        CALL   := CallInstruction;
        OFFSET := PChar(@ThunkHeader) - PChar(@Thunks[i+1]);
      end;
  end;
end;
```

➤ *Above: Listing 7*

➤ *Below: Listing 8*

```
procedure TDll.ActivateThunks;
var i : integer;
begin
  for i := 0 to Count-1 do
    FEntries^[i].Proc^ := @FThunkingCode^.Thunks[i];
end;
```

Then we prepare to call the `TDll` method `DelayLoadFromThunk`. We don't want to write too much assembly code and prefer to handle the more mundane details using Object Pascal. `DelayLoadFromThunk` is a method with one parameter and it uses the default register calling convention:

```
function TDll.DelayLoadFromThunk(
  Thunk: PThunk): pointer;
  register;
```

Because it is a method, it expects to find the `Self`-pointer in the `EAX` register and the `Thunk` parameter in the `EDX` register. Both of these values have previously been pushed to the stack by our generated code thunks (`Self` was pushed explicitly, while the `Thunk` address was pushed implicitly by the `CALL` instruction). We simply copy these values from the stack into their correct register:

```
MOV    EAX, [ESP+12]   // Self
MOV    EDX, [ESP+16]   // Thunk
```

I found the correct locations of these values (relative to the `ESP` register), by analysing the contents of the stack at that point, see Figure 1.

I mentioned previously that we would use the address of the thunk

Continued on page 38

```
procedure ThunkingTarget;
asm
  PUSH    EAX
  PUSH    EDX
  PUSH    ECX
  MOV     EAX, [ESP+12]   // Self
  MOV     EDX, [ESP+16]   // Thunk
  SUB     EDX, TYPE TThunk
  CALL    TDll.DelayLoadFromThunk
  MOV     [ESP+16], EAX
  POP     ECX
  POP     EDX
  POP     EAX
  ADD     ESP,  4
  // "RETurn" to the DLL!
end;
```

➤ *Listing 9*

➤ *Figure 1*

The Case Of The Broken Breakpoints

During the development of the `HVDll` unit presented in this article, I stumbled across a really weird bug in the Delphi IDE's integrated debugger. This incident happened when I was using Delphi 3.02 (build 5.83), but it could also potentially exist in Delphi 4. Unfortunately, I haven't found a step-by-step description of how to recreate this bug, but I wanted to alert you to it so that you can avoid the headaches I had to suffer...

After some initial debugging, my code was working as expected, both in the simple test project provided on the disk and in a much larger real life project at work. The code had been running fine for several days, when I suddenly experienced the dreaded Access Violation exception. I could not for the life of me understand why this was happening but, as the humble programmer I am, I naturally suspected my own code was to blame and busily started debugging.

I quickly found that the first procedural variable referenced in the `Entries` array in the import unit was not being set. It maintained its initial `nil` value. When the code later called through the `nil` pointer, I naturally got the Access Violation exception.

Why wasn't the variable being initialised correctly? It should have been set in the `ActivateMethod`, see Listing 8.

When debugging this code, I found that the value for `FEntries^[0].Proc^` was still `nil` after running the loop! Something was definitely wrong here. I simplified the code as much as possible, but the problem persisted. I used the CPU view to see what was happening at the instruction level.

After simplifying the code, the key assembly instruction for the failing code was:

```
MOV     EDX,    [EDX]
```

This is pretty simple code, right? Not much that could go wrong here. This should simply get the integer pointed to by the `EDX` register and store it back into the `EDX` register. I checked the contents of the address before running this instruction by evaluating:

```
PInteger(EDX)^,X
```

The value at that address was `$005414A0`. It represents the address of `FEntries^[0]`. Stepping over the assembly instruction and then evaluating the `EDX` register, it had somehow become `$005414CC`! This caused the wrong address to be patched. The address was obviously still a pointer to valid writeable memory, because no Access Violation occurred at this stage. Instead, some other global variable was being overwritten. And no change was made to `FEntries^[0].Proc^`, so my procedural variable was still `nil`, causing the access violation when called through later.

I was utterly confused. Further testing showed that the same .EXE file (not recompiled) worked fine when debugged under Turbo Debugger. When running the .EXE file standalone, or turning off integrated debugging in the IDE, it also worked fine. Not a trace of the bug.

I turned on the integrated debugging again and pondered. I tried a build all. It didn't help. I turned off debug info for the unit. That didn't help. I simplified the code. That didn't help. I watched as the code magically performed the invalid lookup, again and again.

Hmmm. The LSB of the integer was consistently being modified from `$A0` to `$CC`. That `$CC` pattern reminded me of something. It is the value I use to fill uninitialized structures in debug mode and it just happens to be the opcode for the `int 3` instruction (breakpoint interrupt). Aha! Just *maybe* the integrated debugger was patching the wrong address when it was setting one of the breakpoints?!

When you create breakpoints in Delphi, the IDE stores the unit name and line number of the breakpoint. When you run the application, Delphi converts this into an address in the code segment. Then it saves the byte that was already there and patches it with the `breakpoint` instruction (op-code `$CC`). What if the line number or converted address is screwed up somehow? What if it doesn't point to an address in your code, but to a global variable?

This is actually what happened in my case. Restarting Delphi didn't help, so I deleted all my breakpoints (some 12 to 15, none of them marked as invalid). Then the code worked again!

I could add new breakpoints and the code still worked.

So, now you have been warned. It seems that the breakpoint information Delphi maintains internally can become corrupt. In my case, it overwrote a global variable. Be aware of this problem if you suddenly find your data overwritten with a `$CC` value during debugging. If the problem goes away when you turn off the integrated debugger, you know what the culprit is. The workaround seems to be to delete all your breakpoints.

If you ever encounter this bug, if you find a consistent way of reproducing it, or if you encounter it in Delphi 4.02, I would very much like to hear about it (as would Inprise).

Continued from page 36

that called us to calculate the index of the DLL routine we are calling. However, the return address pushed by the CPU as part of the CALL instruction in the thunk is not the address of the thunk itself, but rather the address of the instruction following the thunk. To correct for this, we reduce the value of the EDX register with the size of one thunk (ie 5 bytes):

```
SUB     EDX, TYPE TThunk
```

Note that we apparently cannot use the SizeOf operator here: it took me some time to get this right (see *Gotcha! Using SizeOf In BASM* on page 40). Now that we have EAX pointing to the correct TDll instance and EDX pointing to the thunk that called us, we are ready to give control over to the method written in Pascal:

```
CALL    TDll.DelayLoadFromThunk
```

DelayLoadFromThunk is a function, and when it has done its business (more about this later), it will return the address of the actual DLL routine that should be called. If the DLL or routine could not be found, an exception would have been raised, so we don't have to worry about that case. As usual, the function returns its result in the EAX register.

Now we are facing a few tricky issues. Firstly, we need to restore the EAX, EDX and ECX registers to their previous state, but we still have to somehow retain the routine address currently stored in the EAX register. Secondly, we have to restore the state of the stack the way it looked before the thunk was called. Thirdly, we have to somehow transfer control to the DLL routine, without trashing any registers along the way (Object Pascal and general Win32 conventions require us to preserve the EDI, ESI, ESP, EBP and EBX registers).

It turns out we can kill three birds with one stone. The stack has two entries more than it should when entering the DLL routine. In one of these is the address we will return to when leaving. This will always point to the block of thunking code. We don't really want to return to that code, we want to 'return' to the DLL routine instead. This is accomplished by patching the contents of the stack with the address of the DLL routine stored in EAX:

```
MOV     [ESP+16], EAX
```

Now that we don't depend on the value in EAX any more, we are free to restore the three registers to their original (and potentially parameter holding) values:

```
POP     ECX
POP     EDX
POP     EAX
```

Finally, the stack is still skewed by one entry (the value of the Self-pointer), so we simply remove it by adjusting the stack pointer (ESP):

```
ADD     ESP, 4
```

On the top of the stack, we now have the address of the DLL routine. We can transfer control to it by simply returning. The RET instruction has already been added by the compiler as the standard epilogue code of the routine, so we don't have to write any more assembly code:

```
// "RETurn" to the DLL!
end;
```

Voilà! We are now entering the DLL routine and the register contents and stack layout are identical to what they would have been if the user code had called the DLL directly, without going via our thunking routines.

The Order Of Things

Now we have covered all the really tricky parts of the code. There are only a few pieces missing. The actual loading and linking of the DLL routines is taken care of by the TDll class: Listing 10 shows the participating methods.

As we have seen, the assembly code in ThunkingTarget calls into the DelayLoadFromThunk method. This method first calls GetIndexFromThunk to convert the

➤ *Listing 10*

```
function TDll.LoadHandle: HMODULE;
begin
  if FHandle = 0 then begin
    FHandle := Windows.LoadLibrary(PChar(FullPath));
    if FHandle <> 0 then
      Dlls.DllNotify(Self, daLoadedDll, nil);
  end;
  Result := FHandle;
end;
function TDll.GetHandle: HMODULE;
begin
  Result := FHandle;
  if Result = 0 then begin
    Result := LoadHandle;
    if Result = 0 then
      Error(SCannotLoadLibrary,
        [FullPath, SysErrorMessage(GetLastError)]);
  end;
end;
function TDll.LoadProcAddrFromIndex(Index: integer;
  var Addr: pointer): boolean;
begin
  Result := ValidIndex(Index);
  if Result then begin
    Addr := Windows.GetProcAddress(Handle,
      FEntries^[Index].Name);
    Result := Assigned(Addr);
    if Result then
      Dlls.DllNotify(Self, daLinkedRoutine,
        @FEntries^[Index]);
  end;
end;
function TDll.GetProcAddrFromIndex(Index: integer): pointer;
begin
  if not LoadProcAddrFromIndex(Index, Result) then
    Error(SCannotGetProcAddress, [EntryToString(
      FEntries^[Index]), FullPath, SysErrorMessage(
      GetLastError)]);
end;
function TDll.DelayLoadIndex(Index: integer): pointer;
begin
  Result := GetProcAddrFromIndex(Index);
  FEntries^[Index].Proc^ := Result;
end;
function TDll.GetIndexFromThunk(Thunk: PThunk): integer;
begin
  Result := (PChar(Thunk) - PChar(@FThunkingCode^.Thunks
    [0])) div SizeOf(TThunk);
end;
function TDll.DelayLoadFromThunk(Thunk: PThunk):
  pointer; register;
begin
  Result := DelayLoadIndex(GetIndexFromThunk(Thunk));
end;
```

Thunk address to a routine index and then calls DelayLoadIndex to find and return the address of the corresponding DLL routine.

In GetIndexFromThunk, we calculate the routine index by subtracting the start of the array from the thunk address, finally dividing by the size of a single thunk. We have now found the index of the thunk that called us in the FThunkingCode^.Thunks array. The same index can be used in the FEntries^ array to get the address of the procedural variable and the name or ordinal of the DLL routine. The index we just calculated is passed on to DelayLoadIndex. This first calls GetProcAddrFromIndex to find and return the DLL routine, then patches the procedural variable so that it points to the address found. If the DLL or routine could not be found, an exception is raised, so we don't have to add any error checking at this level.

Following the call chain, we now look at GetProcAddrFromIndex. This aptly named routine passes the buck on to LoadProcAddrFromIndex. If the routine could not be found, we raise an exception by calling the class method Error.

Time to get some work done, don't you think? Well, your patience has been rewarded. We are now in LoadProcAddrFromIndex. This routine tries to get the address of the DLL routine, if it cannot it returns false. First it checks to see if the index parameter is valid. Then it calls the Win32 API GetProcAddress to get the address of the routine at that entry. If we receive a non-nil value, we call the notification method in the DLLs object.

In the call to GetProcAddress, we reference the Handle property. This property has a GetHandle read access method. In GetHandle, we check to see if a valid handle already exists. If not we call the LoadHandle method to get it and raise an exception via the Error method if it still hasn't been loaded.

If necessary, LoadHandle calls the Win32 API LoadLibrary to actually load the DLL into memory. If the DLL loaded successfully, we again notify about this using the DllNotify method of the DLLs object.

Yawn... Are you still awake? That description might sound like we are doing things in a very roundabout way, but the reason I have divided it up this much is to have building blocks for additional functionality, as we will soon see.

```
TDll = class(TObject)
  public
    constructor Create(const DllName: string; const Entries: array of TEntry);
    destructor Destroy; override;
    procedure Load;
    procedure Unload;
    function HasRoutine(Proc: PPointer): boolean;
    function HookRoutine(Proc: PPointer; HookProc: Pointer;
      var OrgProc): boolean;
    function UnHookRoutine(Proc: PPointer; var OrgProc): boolean;
    property FullPath: string read FFullPath write SetFullPath;
    property Handle: HMODULE read GetHandle;
    property Loaded: boolean read GetLoaded;
    property Available: boolean read GetAvailable;
    property Count: integer read FCount;
    property EntryName[Index: integer]: string read GetEntryName;
  end;
```

➤ *Listing 11*

Using The Classes

We have now been through the inner workings of the HVDll unit. What might be more useful in the long run is to know how the classes can be used for everyday work. I have included the public interface of the TDll class in Listing 11.

TDll is the class you should be instantiating in your import unit. To give the application developer greater flexibility and control, you also want to put a reference to this instance in the interface of the unit. We saw an example of this and how to call the Create constructor back in Listing 5. If you feel like it, you can free your TDll instance in your finalization section, but this will be done automatically by the Dlls object anyway.

You can use the Load and Unload methods to have better control on when the DLL is loaded and released from memory. Note that Load will raise an EDllError if it has problems linking to any of the routines. Use the Loaded property to check if the DLL is already in memory and the Available property to see if the DLL can be loaded (this will not raise any exceptions). The Handle routine returns the module handle of the DLL. Again this raises an exception if it could not be loaded.

Use HasRoutine to see if a specific routine exists in the DLL. This is done by passing in the address of the procedural variable you want

Gotcha! Using SizeOf In BASM

I found a minor quirk in the way that Delphi's built-in assembler (BASM) works. If you use the SizeOf operator inside some assembly code, it will happily compile it for you, but the result is not what you might expect. For instance, compiling the following assembly statement:

```
MOV     EAX, SizeOf(byte)
```

you would expect the compiler to produce:

```
MOV     EAX, $1
```

However, the code actually produced is:

```
MOV     EAX, $34
```

As you can see, the BASM compiler evaluates SizeOf(byte) to $34 (or 52 decimal)! In fact, using the SizeOf operator on any structure, no matter how large it is, it will always evaluate to 52! I have no idea why this happens, but the compiler does not generate any error or warning, so you should be aware of it.

There are two possible workarounds. First, you could simply use a Pascal constant that has been assigned to the correct SizeOf expression:

```
const
  ByteSize = SizeOf(byte)
asm
  MOV     EAX, ByteSize
```

Even better, you can use the BASM operator designed for the job, the TYPE operator:

```
MOV     EAX, TYPE byte
```

In the context of the SizeOf operator, it seems the answer to life is not 42 any more, according to BASM it's 52!

Calling Performance And Package Overhead

Originally, I had intended to blatantly announce that the technique of using procedural variables to call DLL routines actually produces slightly faster code than calling the same routines using implicit loading. After further investigation, I found that this is actually the case for code compiled with Delphi 2, while the situation is rather more complex in Delphi 3 and 4, due to the package support these compilers must adhere to.

For implicitly loaded routines, the compiler CALLs an address that again JMPs via a global variable to the actual DLL routine. So there are three levels of indirection, see Figure 1 in this boxout.

In Delphi 2, when calling through a procedural variable, there are only two levels of indirection, it CALLs directly via the procedural variable. See Figure 2 in this boxout.

One Step Backwards

We have now seen that Delphi 2 had a very efficient implementation of calling routines through procedural variables. I found that the same code produced more complex (and thus slower) code in Delphi 3 and 4. See Figure 3 below.

Here we are back to three levels of indirection. The instruction pointer only changes once, so this code might still be marginally faster than calling implicitly loaded routines. However, instead of loading the contents of the global variable directly, it first gets the address of that global variable from some other automatically created global variable, called EntryP in the illustration. Somehow EntryP always contains the address of the variable we're interested in, Entry.

Why is this? Why not encode the address of the global variable directly into the generated assembly code? The short answer is package support.

Package Support Bites Back!

The reason for this extra level of indirection is to support packages. Whenever you use any global variable from another unit, the compiler will generate code that gets the actual address of that variable from another, automatically created global variable. The reason is that if the unit you used happens to reside, not in your .EXE file, but in an external package, the address of that global variable will be fixed up when the .EXE starts and loads the package. This gives great flexibility, at the cost of producing slightly larger and slower code.

The problem is that this overhead is present even if your application does not use packages at all. This is to assure that pre-compiled DCUs must not be recompiled if you decide to use packages.

I have previously requested Inprise to document how the package support in Delphi is implemented. This kind of information can be important when debugging. So far, they have declined to do it.

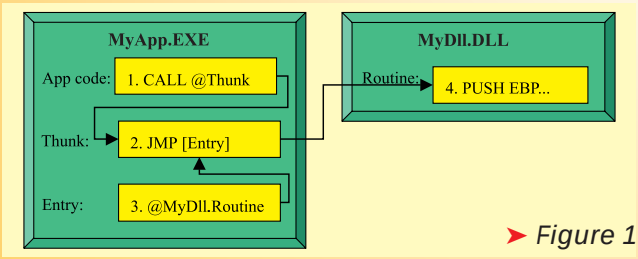


Figure 1

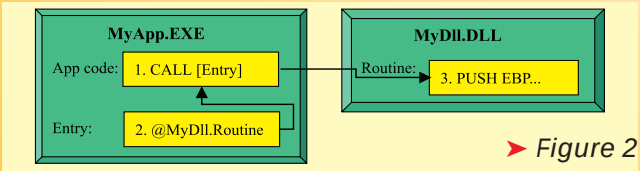


Figure 2

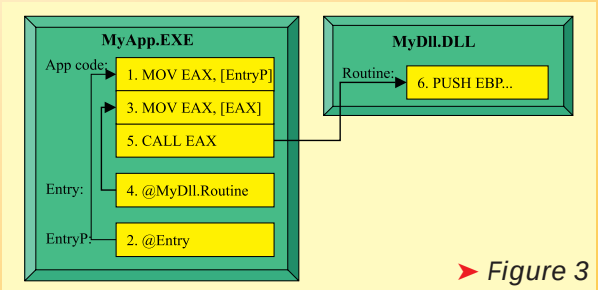


Figure 3

to check. This can be used to disable features in the application that depend on certain routines to be available in the DLL, like this:

```
NewFeatureMenuItem.Enabled :=
  MyDll.HasRoutine(
    @@NewFeature);
```

The FullPath property is initially set to the DllName parameter of the constructor, but you can change this at runtime to load DLLs in specific directories or with different names. Note that if you don't specify a specific directory, the standard Windows searching strategy will be used (see the help file on LoadLibrary). If you change this property after the DLL has already been loaded, the DLL will be unloaded and all the thunks reactivated. The next time a routine is needed, the new DLL will be loaded and linked to.

To get the name of a specific import entry, use the EntryName indexed property. The Count property will return the number of import entries that have been registered for this TDll object. These properties can be useful inside a handler for the OnDllNotify event of the Dlls object (see below).

Finally, there are two methods to support hooking of the DLL routines. This is mainly useful for debugging purposes. Call HookRoutine to hook a specific DLL routine by giving the address of the procedural variable, the address of your hook routine and another procedural variable to hold the previous value (the back-hook). Be careful to ensure that both your hook routine and the procedural variable use exactly the same parameters and calling convention as the original DLL routine. To turn off hooking, use the UnHookRoutine method.

There is also a class that maintains a list of all TDll instances that have been created. You can see the public interface of this class in Listing 12.

The TDlls class holds a reference to all TDll objects created in

Continued on page 44

Proper Code Generation

In the article, we look at how to create code on-the-fly to be executed. However, we didn't talk about the pre-requisites for this to work. We cannot just use any old memory block allocated with GetMem, for instance. We would be able to patch in the code instructions alright, but upon executing the code, we would get an Access Violation.

The reason for this is that all memory pages in the system have certain access rights associated with them. By default, memory allocated with Delphi's GetMem procedure cannot be executed, because the memory pages only have PAGE_READWRITE and not the PAGE_EXECUTE right.

To avoid this problem we must explicitly set the desired access rights for the memory block. This can be done with the VirtualProtect routine. With this knowledge in hand, we could be tempted to write code such as this:

```
System.GetMem(FThunks, 100);
// Patch in the code instructions here...
VirtualProtect(FThunks, 100, PAGE_EXECUTE,
  @OldProtection);
```

Here we get a memory block using GetMem. This block now has READWRITE rights. We patch in all the required dynamic code instructions. Then we change the rights of the memory block so that it can be executed.

Now, this code would work initially. You would be able to call the generated code without causing Access Violations. However, after a short time you would probably experience strange Access Violations inside Delphi's memory manager. The reason is that the VirtualProtect routine doesn't only change the rights of the 100 bytes of memory that we requested. It is not that granular. Instead it will change the rights in steps of 4Kb pages. This means that other blocks of memory used by Delphi's memory manager also get their rights changed from READWRITE to EXECUTE only. When Delphi subsequently tries to read or write to those memory blocks, the Access Violation is generated.

Hmm. How do we get around this limitation? Well, looking at the documentation for VirtualProtect, it is apparent that there is a combination of rights that would be right (*sic*) for us. Let's try to change the code so that it uses PAGE_EXECUTE_READWRITE instead.

Well, now the code finally works. But the question remains: what should we do when we are going to free the memory block? We could leave the rights as they are, but that would be quite messy. After some time, large amounts of the heap memory would have EXECUTE rights and any invalid jumps into memory could go undetected, causing all kinds of problems. Let's set the rights back before releasing the memory:

```
VirtualProtect(FThunks, 100, PAGE_READWRITE,
 @OldProtection);
System.FreeMem(FThunks);
```

That would clean up properly for this block. However (how many howevers can there be?), this could potentially change the rights of *another* block of code from EXECUTE_READWRITE back to READWRITE if it just happens to be in the same memory page as FThunks. We could try to get around this by keeping the value of the OldProtection variable when allocating the memory and using this to set the rights back to the previous state. This would solve some cases, but not when the allocation and deallocation order differs.

Oh... Sigh! We seem to have stepped into a wormhole here *[Wow! Time travel in Delphi! Ed]*. Let's take a step backwards to see what we are doing wrong. The problem with using the GetMem/VirtualProtect combination is that GetMem has a granularity of 4 bytes, while VirtualProtect has a granularity of 4Kb. So we have to forget using GetMem for this purpose.

What other memory allocation routines are available? Well, we have the VirtualAlloc routine. This will allocate blocks in 4Kb page chunks and it will even let us set the rights of the memory pages directly. Perfect! Using VirtualAlloc directly is actually a very good solution. We would do something like this:

```
FThunks := VirtualAlloc(nil, 100, MEM_COMMIT or
  MEM_RESERVE, PAGE_EXECUTE_READWRITE);
```

After this allocation (assuming it succeeded), we would be able to patch in the code and execute it without any problems and there would be no conflicts with other code sharing our page of memory. However (here we go again), the fact that VirtualAlloc rounds the allocation up to the nearest 4Kb boundary is also its greatest problem. I started my programming career using a ZX-81 with the incredible amount of 16Kb of memory, so the thought of wasting 4Kb to get 100 bytes of memory makes me twitch. There must be a better way.

What we are looking for is the combination of GetMem's granularity and VirtualAllocs privacy when it comes to page rights. What we want is a private heap just for generating code on the fly. This heap could be shared by all entities that need memory blocks with EXECUTE_READWRITE rights. Well, luckily there is a set of routines that makes it a breeze to implement private heaps: HeapCreate, HeapAlloc, HeapSize, HeapFree and HeapDestroy. There are other routines as well, but these are the ones we need for our simple purpose.

To avoid using these arcane Win32 routines directly, I've implemented a couple of wrapper classes in the HVHeaps unit (see the code on the disk). The TPrivateHeap class is a generic wrapper around the heap routines and it provides the interface in Listing 1 in this boxout.

This class will allocate memory blocks with the normal READWRITE rights. In normal situations, you would simply use the GetMem and FreeMem methods that work just like their equivalents in the System unit, only that the heap they allocate from is private.

This class can be useful in itself to reduce memory fragmentation. If you have two large, independent subsystems in your application that both allocate large numbers of memory blocks, you could improve the memory utilisation and potentially performance by using separate private heaps for each subsystem. To allocate object instances from the private heap, you could override `NewInstance` and `FreeInstance` and let the private heap object take care of the allocations (see my article *The Rise And Fall Of TObject* in Issue 35 for more details of this technique).

For our specific purpose of allocation memory blocks that can be executed, I have defined the `TCodeHeap` class, see Listing 2.

Notice that the `GetMem` method defined in `TPrivateHeap` was virtual. Here we override it, call the inherited `GetMem` to do the actual allocation and then immediately set the page protection flags to `EXECUTE_READWRITE`. We are guaranteed that all the memory pages belong to this private heap, so there will be no conflict in doing this. We might set the rights of existing memory blocks within the heap, but that does not matter as they would already have been set to `EXECUTE_READWRITE`. All users of the code heap agree that the rights should be the same, so there is no danger of conflict.

In the `HVDll` unit, an instance of `TCodeHeap` is created and stored in a field in the `Dlls` instance. In the `CreateThunks` method of `TDll`, we use the `GetMem` method of this code heap to do the allocations. This is a clean and efficient way of getting memory blocks that can be used for dynamic code generation.

➤ *Listing 1*

```
type
  TPrivateHeap = class(TObject)
  public
    destructor Destroy; override;
    procedure GetMem(var P{: pointer}; Size: DWORD);
      virtual;
    procedure FreeMem(P: pointer);
    function SizeOfMem(P: pointer): DWORD;
    property Handle: THandle read GetHandle;
    property AllocationFlags: DWORD
      read FAllocationFlags write FAllocationFlags;
  end;
```

➤ *Listing 2*

```
TCodeHeap = class(TPrivateHeap)
  public
    procedure GetMem(var P{: pointer}; Size: DWORD);
      override;
  end;
implementation
procedure TCodeHeap.GetMem(var P{: pointer};
  Size: DWORD);
var
  Dummy: DWORD;
begin
  inherited GetMem(P, Size);
  Win32Check(Windows.VirtualProtect(Pointer(P), Size,
    PAGE_EXECUTE_READWRITE, @Dummy));
end;
```

Continued from page 41

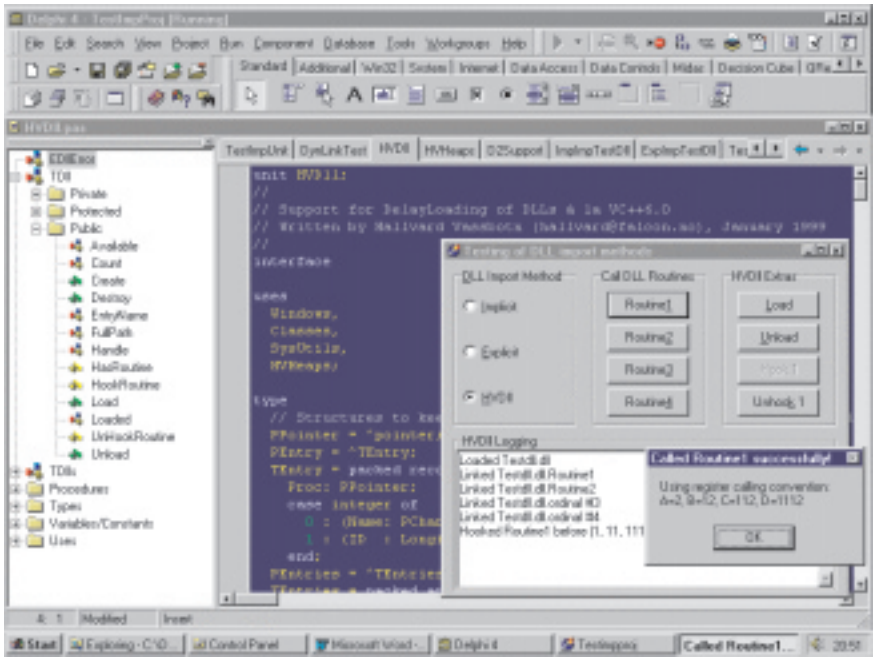
the application. Early in the project this class was central when converting a TThunk address to a TDll instance and routine index combination. Basically, a method of the TDlls class would scan through all the TDll objects to find the object that owned the thunk. With the improved thunking I finally implemented (with the per-DLL thunk in addition to the per-routine thunks), this logic was no longer necessary.

I decided to keep the TDlls class and the global Dlls instance, anyway. It provides a central point of access to all dynamic DLL objects in the application. It automatically frees remaining TDll instances when the application closes. Finally, it allows you to implement a handler of the OnDllNotify event. This event is called whenever a DLL is loaded or unloaded, or when a DLL routine is first linked to.

Demonstration Project

As usual, there is the traditional demo project, see Figure 2. It uses three different import units to access the routines in a DLL called Testdll.Dll. You might think this is overkill, but we only want to demonstrate three ways of importing DLL routines. You can select what import method to use by selecting in the radiogroup labelled DLL Import Method. The Testdll.dll exports four routines taking four parameters, but each with a different calling convention. This is so we can easily check that the import method we are using works properly with all combinations. If you select the HVDll method you will also see some messages in the logging pane, notifying when the DLL is first loaded, when the routines are linked to and when the DLL is being unloaded.

➤ *Figure 2*



```
TDllNotifyAction = (daLoadedDll, daUnloadedDll, daLinkedRoutine);
TDllNotifyEvent = procedure(Sender: TDll; Action: TDllNotifyAction;
  Index: integer) of object;
TDlls = class(TList)
  public
    constructor Create;
    destructor Destroy; override;
    property Dlls[Index: integer]: TDll read GetDlls; default;
    property OnDllNotify: TDllNotifyEvent read FOnDllNotify write FOnDllNotify;
  end;
```

➤ *Listing 12*

Comparing With /DELAYLOAD

The HVDll unit gives Delphi users many of the same benefits as the /DELAYLOAD option of the VC++ 6.0 linker. However, there are a number of differences. This has mainly to do with the fact that /DELAYLOAD is implemented as part of the linker, while HVDll is implemented as just another unit that you include in your import units.

Because of the linker support, VC++ users simply have to add an option to the already confusing array of project settings. No code changes whatsoever are needed.

Contrast this with the HVDll solution, which forces you to write an explicit import unit following the format in Listing 3, and then use this unit instead of the implied import unit. To make this easier, it would be possible to write a small conversion utility to automatically turn existing import units into dual-mode import units that could implement implicit or explicit linking according to a compiler DEFINE.

On the positive side, the HVDll solution generates smaller per-DLL and per-routine thunks. It is also interesting to note that the VC++ thunks do not preserve the contents of the EAX register. I initially thought that this was a major shortcoming in that it would not support the register calling convention. However, it turns out that VC++'s __fastcall calling convention only uses EDX and ECX to pass parameters, so it is safe to trash EAX. It just means that Delphi DLLs using the register calling convention cannot be easily used by VC++ (and that it would fail utterly if you tried to /DELAYLOAD such a DLL).

Furthermore, the HVDll solution is object oriented, so it is easy to extend. You can easily set the path and name of the DLL at runtime, and you have the complete source code. And last, but not least, it works with Delphi!

Acknowledgement

Peter Sawatzki (www.sawatzki.de/default.htm) wrote a similar 16-bit unit for BP and Delphi 1.0. His code inspired me to develop and extend the idea on the Win32 platform.

Hallvard Vassbotn is a Senior Software Developer at Reuters Norge AS, Falcon R&D. You can reach him at hallvard@balder.no