# Delphi Memory Consumption

*by Brian Long*

This article looks into a possible problem with Delphi and small API utility applications, and also investigates how to calculate the amount of memory a Delphi application is using at any given time.

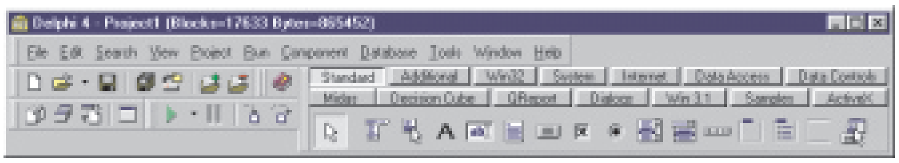
To give a context for the article, it is useful to see a question that was recently sent to the *Delphi Clinic*:

*'I am writing an API-only utility in Delphi 3, which I have working in an EXE of only 20Kb. However, its memory footprint is 1,220Kb! The EXE is only using API calls, huge strings and* `PChars`*, but it insists on linking in OLE32.DLL and OLEAUT32.DLL. After much investigation, it appears the reason is a call to a* `Variant` *clear function (*`VariantClear`*) which comes from OLEAUT32.DLL. The only reason I can see for this being included is two variables declared in the* `System` *unit of type* `Variant` *(*`Null` *and* `Unassigned`*). Is there any way of creating a D3/D4 app without these DLLs being statically linked?'*

I've seen this question a couple of times now, both on a newsgroup and mailed to me. It challenges the premise that Delphi is just as good for writing API-level utilities as it is for writing more complete VCL form-based applications. I wonder what the answer will be...

➤ Listing 1

```
function CommittedMemorySize: DWord;
var
  MBI: TMemoryBasicInformation;
  SI: TSystemInfo;
  RangeStart: Pointer;
begin
  Result := 0;
  GetSystemInfo(SI);
  RangeStart := SI.lpMinimumApplicationAddress;
  while DWord(RangeStart) < DWord(SI.lpMaximumApplicationAddress) do begin
    VirtualQuery(RangeStart, MBI, SizeOf(MBI));
    //Only get committed memory (storage allocated for this)
    if MBI.State = MEM_COMMIT then
      Inc(Result, MBI.RegionSize);
    //Delphi 2 & 3 could only handle $7FFFFFFF as biggest integral number
    //Last region is likely to end at $80000000. To avoid integer
    //overflow, we'll do a comparison and bypass the addition
    if DWord(SI.lpMaximumApplicationAddress)-MBI.RegionSize
      >= DWord(RangeStart) then
      Inc(PChar(RangeStart), MBI.RegionSize)
    else
      //If overflow would have occurred, loop is over
      Break
  end;
end;
```


Delphi 4 - Project1 [Blocks=17633 Bytes=865452]

➤ *Figure 1: The -HM command-line switch causes Delphi 4 to display memory usage.*

First of all, we need an API-only application that makes use of no VCL units, to test the problem with (provided as APIProg.Dpr on this month's disk). Having got that, we then need to identify how much memory a Delphi application is using at any given time. Of course, if you are running under Windows NT, this is easy: the Task Manager has a page of running processes that displays the size of the process address space. But what about Windows 95 and 98?

## How Much Memory Are We Using?

Before investigating the original Delphi Clinic question, we will go off on a digression.

The answer to the question *'How much memory is my program using at runtime?'* has various answers. You might mean *'How many bytes of memory has my program allocated for its various needs with* `GetMem`, `AllocMem` *etc, and also with various object constructors being called?'* But you might also mean *'How much storage has Windows set aside for my application's code and data requirements?'*

With Windows NT in the equation, you may also be referring to your application's working set size (a Windows NT concept not supported by Windows 95/98). This last option can be examined using the `GetProcessWorkingSetSize` NT API (not supported under Windows 95/98). For a more generic solution, let's look at the first two ideas.

## API Analysis

Win32 has a `VirtualQuery` API. Given an address, it will give you information on the block of memory containing that address, including the block's size and whether Windows has allocated any storage for it.

One approach would be to start at the beginning of your process's accessible address space, and call `VirtualQuery` to see if that block has been allocated to you. Then increment the address by the size of that block, and do it again. This can be continued until you reach the end of your address range. The accessible address range can be obtained through a call to the `GetSystemInfo` API. You could make up a function like the one in Listing 1 to work this out.

Notice that the code caters for limits in Delphi 2 and 3, which cannot handle integer values greater than `$7FFFFFFF`. Since the final memory region examined will more than likely cause `RangeStart` to be given a value of `$80000000`, causing an integer overflow (which I found, to my chagrin), the code checks if the `while` loop condition would fail with a subtraction rather than an addition. If it would, the loop is explicitly terminated.

Listing 1 checks only for *committed* memory. A page of committed memory is one that has physical storage allocated for it, either in memory or on disk. Apart from a *free* page, which is self-explanatory, the only other type of memory page is a *reserved* page. A reserved page is a memory page from the process's virtual address space that has been set aside for future use, but for which no physical storage has yet been allocated. Checking only for committed memory means that we are totting up the amount of allocated storage for a program, for all the code loaded from the EXE and DLLs, as well as all their data.

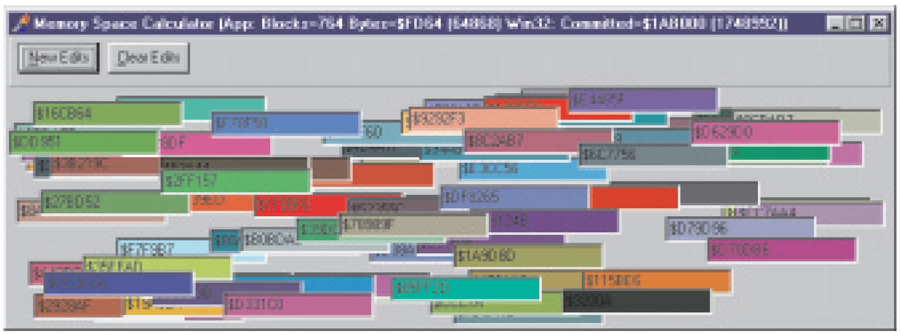
## RTL Analysis

So that caters for calculating *all* the memory Windows has allocated for your process. Now how about the memory your process has actively requested?

This is very easy in Delphi 2 or higher, as the `System` unit has two variables that we can use. `AllocMemCount` indicates the number of requested memory allocations that have yet to be freed and `AllocMemSize` tells you how many bytes are currently allocated by your program.

In a normal Delphi application, `AllocMemSize` will probably keep growing in fits and starts, due to the VCL allocating internal helper objects and heap-based records, as the application tries to do various internal housekeeping. Here and there it will also decrease. If you see a constant rise in this value, then it is time to worry about your code causing memory leaks.

➤ *Figure 2*


Memory Space Calculator [App: Blocks=764 Bytes=$FD64 (64868) Win32: Committed=$1AB000 (1748992)]

```
procedure TForm1.FormCreate(Sender: TObject);
  function FindHMCommand: Boolean;
 {$ifdef DelphiLessThan4}
  var I: Integer;
  begin
    Result := False;
    for I := 1 to ParamCount do
      if (UpperCase(ParamStr(I))='/HM') or (UpperCase(ParamStr(I))='-HM') then
        Result := True;
 {$else}
  begin
    Result := FindCmdLineSwitch('HM', ['/', '-'], True)
 {$endif}
  end;
begin
  if FindHMCommand then begin
    tmrHeapMonitor.Enabled := True;          //Enable heap monitoring (via timer)
    tmrHeapMonitor.OnTimer(tmrHeapMonitor);  //Make the timer tick straight away
  end;
  EditList := TList.Create;
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  EditList.Free
end;
procedure TForm1.tmrHeapMonitorTimer(Sender: TObject);
begin
  Application.MainForm.Caption := Format(
    '%s (App: Blocks=%d Bytes=$%x (%2:d) Win32: Committed=$%x (%3:d))',
    [Application.Title, AllocMemCount, AllocMemSize, CommittedMemorySize]);
end;
```

➤ *Listing 2*

A word of warning regarding the use of `AllocMemSize` and `AllocMemCount` is warranted here. If you are writing an application, and also one or more DLLs to go with it, and you have decided to use the shared memory manager (by adding `DelphiMM` to your `uses` clause), then these variables should be avoided. Instead you should call `GetAllocMemCount` and `GetAllocMemSize` from the `DelphiMM` unit.

The Delphi 4 (and later) IDE support a command-line switch `-HM` (or `/HM`) that causes memory size information to be added to the main IDE window's caption (see Figure 1). This switch was undocumented in Delphi 4 but has been documented for version 5.

We can do exactly the same thing in any of our programs by enabling a timer if the command-line is present. The timer can read from the relevant variables (or call the functions) and modify the caption bar. To find a command-line switch, users of Delphi 4 (and higher) can use the routine `FindCmdLineSwitch`. Earlier versions have to find it manually.

Figure 2 shows a sample program (MemUse.Dpr on this month's disk) started with a `-HM` command-line. Listing 2 has the pertinent code. The program has a couple of buttons that respectively create a bunch of edit controls, adding them to a list, and destroy them, so you can see the memory count fluctuating. With the edit controls visible in the screen shot, Windows has allocated `$1AB,000` bytes for my application's total requirements. The program itself has 764 memory allocations still in use, totalling 64,868 bytes.

As well as the `-HM` command-line switch, Delphi 4 also introduced `-HV` or `/HV`. This causes Delphi to keep checking the status of its heap for errors. If an error is found, the error code is written on the main window's caption bar. If we wish, we can also replicate this behaviour by repetitively calling the `GetHeapStatus` function from the `System` or `DelphiMM` unit. Listing 3 shows a suitable routine, using textual descriptions as well as the heap error code number for (an arguable amount of) clarity.

Having checked the complex memory management code in GetMem.Inc, it seems that `GetHeapStatus` will only report a subset of errors. Many of the helper routines set a variable to indicate the current error status. GetHeapStatus immediately overwrites this variable with cHeapOk on entry. Consequently, you are only likely to see cBadCurAlloc, cBadUsedBlock, cBadNextBlock, cBadFreeList, cBadFreeBlock and cBadBalance. If you have Turbo Assembler, the RTL source, and some nerve, you could fix the problem by commenting out the offending line and recompiling the RTL and VCL source with the Inprise-supplied makefile. This would only help in the case of standalone applications: the Inprise-supplied runtime packages cannot be recompiled.

The THeapStatus record returned by GetHeapStatus has various fields that give more details on the quantity of memory allocated by your application, all of which are described in the online help.

## Back To The Story

Now that we have a couple of ways of seeing how much memory our program uses, we can get back to the original question. You may recall we needed an API-only application to test the problem with. As mentioned above, this month's disk has a project called APIProg.Dpr that will do. It uses Windows API calls to create a main window, and has a menu that allows you to display an *About* box, and draw coloured lines on the main window. In addition, it employs Windows timer messages to constantly write on the caption bar the amount of committed memory Windows has set aside for this application (using code from Listing 1).

The details of how the application works are beyond the scope of this article, and for many people are irrelevant, but it shows the amount of work required to get a window and a dialog on the screen without the help of a framework like the VCL. Windows are implemented solely in code. Dialogs and menus have to have resources defined and linked into the program. Lots of messages have to be handled to get things working.

When the application is launched in Delphi 2, it claims to take up 88Kb. Dropping down any menu causes this to increase to 92Kb, but either way, I'm sure you will agree this is an acceptably low overhead.

## Memory Hog

If you now launch the application in Delphi 3 or later, the application reports 1,348Kb of committed memory! This figure comes up on Windows 95/98. Windows NT causes the program to report 4,684Kb and the NT task manager reports 1,148Kb. I haven't looked into what type of memory blocks the task manager tots up to calculate its figure. The reason for this was explained in the question, but let's take it step by step anyway, to see how it was worked out.

Firstly, you should run the application through the command-line tool TDump (included with Delphi). This will tell you all the routines, from various DLLs, that your application is calling. Delphi 2 and 3's versions of TDump generate large amounts of information, including the import table that contains the details we need. Delphi 4 adds an -em command line switch to list the import table only. From the list that comes out, we can ignore anything that comes from the standard Windows DLLs: KERNEL32.DLL, GDI32.DLL and USER32.DLL.

The Delphi 2 application has no imports other than from the standard Windows DLLs. The Delphi 3 (and later) applications do: RegOpenKeyExA, RegQueryValueExA and RegCloseKey, all from ADVAPI32.DLL, along with VariantClear from OLEAUT32.DLL.

So now we need to examine why they are used, and whether their use is really necessary. Figure 3 shows the list of modules in the program's memory space (Delphi 3's View | Modules, or View | Debug Windows | Modules in Delphi 4 or later). You can see the unwanted DLLs, along with another one: OLE32.DLL. This is probably implicitly loaded by OLEAUT32.DLL.

## Do We Need The Registry?

Let's take the registry routines from ADVAPI32.DLL first. These are used twice. Firstly, if your program is running on a Japanese NEC machine, code in the System unit checks to see if the

```
HKEY_LOCAL_MACHINE\SOFTWARE\
  Borland\Delphi\
  RTL\FPUMaskValue
```

registry value exists. If it does, the value is used to customise the lower ten bits of the floating point unit (FPU) control register. This

➤ *Listing 3*

```
function HeapErrorDesc(Error: Cardinal): String;
begin
  //Error code constants come from Delphi's Source\RTL\GetMem.inc file
  case Error of
    cHeapOk:            Result := 'Everything''s fine';
    cReleaseErr:        Result := 'OS returned an error when we released';
    cDecommitErr:       Result := 'OS returned an error when we decommited';
    cBadCommittedList:  Result := 'List of committed blocks looks bad';
    cBadFiller1,
    cBadFiller2,
    cBadFiller3:        Result := 'Filler block is bad';
    cBadCurAlloc:       Result := 'Current allocation zone is bad';
    cCantInit:          Result := 'Couldn''t initialise';
    cBadUsedBlock:      Result := 'Used block looks bad';
    cBadPrevBlock:      Result := 'Prev block before a used block is bad';
    cBadNextBlock:      Result := 'Next block after a used block is bad';
    cBadFreeList:       Result := 'Free list is bad';
    cBadFreeBlock:      Result := 'Free block is bad';
    cBadBalance:        Result := 'Free list doesn''t match blocks marked free';
  end
end;
procedure TForm1.tmrHeapMonitorTimer(Sender: TObject);
var
  HS: THeapStatus;
begin
  HS := GetHeapStatus;
  if HS.HeapErrorCode = cHeapOk then
    Application.MainForm.Caption := Format(
      '%s (App: Blocks=%d Bytes=$%x (%2:d) Win32: Committed=$%x (%3:d))',
      [Application.Title, AllocMemCount, AllocMemSize, CommittedMemorySize])
  else
    Application.MainForm.Caption := Format('%s (Invalid heap code %d: %s)',
      [Application.Title, HS.HeapErrorCode, HeapErrorDesc(HS.HeapErrorCode)])
end;
```

control register value is stored in the System unit variable Default8087CW.

Also, the SysInit unit (implicitly used in an application written in Delphi 3 or later, much like the System unit) tries to find Delphi resource modules by calling LoadResourceModule. This System unit function looks for special DLLs with a name the same as the application, and an extension matching the current Windows locale three-letter abbreviated language name (ENG for British English). This would be a language and country translation of the application's form and string resources. Then it looks for a file with just the first two out of these three file extension letters (EN for English). This would be a language translation, independent of country.

Before it tries either of these, it checks in the

```
HKEY_CURRENT_USER\Software\
  Borland\Delphi\Locales
```

and

```
HKEY_CURRENT_USER\Software\
  Borland\Locales
```

registry keys to see if there are application-specific file extension overrides it should be using.

So on a non-Japanese machine, the registry will be implicitly accessed, regardless of our desires. You could remove ADVAPI32.DLL from memory some time after the program starts up but, having tested this, you gain nothing by doing so. The memory consumption does not drop at all. Evidently, ADVAPI32.DLL does not consume any working space for itself. In any case, even if you do remove it ADVAPI32.DLL could well be used implicitly by your application anyway, thanks to USER32.DLL importing some of its routines (running TDump across USER32.DLL would verify this). If this happened after your removing it, you would get Access Violations which would be very hard to track down.

## Do We Need OLE Calls?

Now, onto OLEAUT32.DLL. The only call the application makes to this DLL is VariantClear (highlighted in Figure 3). This API is only called from within a System unit function VarClear, actually implemented as _VarClear. VarClear is not called by the code in the project, but the System unit calls it from many Variant support routines. It also calls it from _VarClr, a routine that is automatically called by compiler-generated code to tidy up Variant variables when they go out of scope.

Since there is no explicit Variant manipulation in the program, I agree that the compiler is calling _VarClr to tidy up the two Variant variables defined in the System unit, when the program is closing down. In Delphi 3 and later, these two Variants, Null and Unassigned (see online help for more details about them), are accompanied by another one, EmptyParam. This means that two or three calls to _VarClr at the end of the program cause calls to VarClear, which in turn cause calls to VariantClear.

Delphi 2 does not appear to clear up these global Variant variables automatically, which is why the memory consumption is so small. However, if you use the SysUtils unit, Delphi 2 apps will pull in the DLL, as explained by Hallvard Vassbotn in Issue 14's *Tips & Tricks* column.

In the case of Delphi 3 and later, we can do something about these calls. The implementation of VarClear is in assembler, but in summary, it will not call VariantClear (from the DLL) if the Variant is marked as empty. So now we have all that we need to solve the problem. The steps will be as follows. Firstly, set the System Variant variables so they are marked as empty. Then unload the OLEAUT32.DLL from memory.

The APIProg.Dpr project has all the relevant code in it to accomplish the goal, or at least get as close to it as we can. The question asked if we can avoid statically linking to the unwanted DLLs. Well no, but we can unload them if we know they won't be used again. Listing 4 shows a procedure that deals with the memory overhead in an API-oriented application.

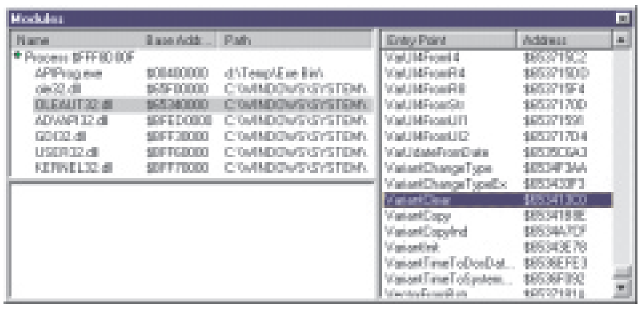
Note that the sample project *only* calls this routine if you pass a command-line parameter (which can be anything you like). Use Run | Parameters to set one up. Also, any subsequent use of any Variant variables or parameters will cause Access Violations, as the important support library has been unloaded.

Null and Unassigned caused me a problem for a while. Although they are declared in the System unit as variables, we are unable to modify them. The compiler rejects any attempt to do so and treats them

```
procedure ReduceMemoryOverhead;
begin
  //Stop the RTL wanting to clear System Variants
 {$ifndef DelphiLessThan4}
  TVarData(EmptyParam).VType := varEmpty;
 {$endif}
  //These two seem to be considered constants, so we hack around
  //this by de-referencing the "constant" item's address
  TVarData((@Null)^).VType := varEmpty;
  TVarData((@Unassigned)^).VType := varEmpty;
  //Unload OLEAUT32.DLL, which will in turn unload OLE32.DLL
  FreeLibrary(GetModuleHandle('OLEAUT32.DLL'));
end;
...
if ParamCount > 0 then
  ReduceMemoryOverhead;
```

➤ *Listing 4*

➤ *Figure 3*



```
Modules
Name            Base Addr   Path                   Entry Point            Address
Process $FFF80 00F                                 VarUI4FromI4           $653719C2
  APIProg.exe   $00400000   d:\Temp\Exe Bin\       VarUI4FromR4           $653719DD
  ole32.dll     $65F00000   C:\WINDOWS\SYSTEM\     VarUI4FromR8           $653719F4
  OLEAUT32.dll  $65340000   C:\WINDOWS\SYSTEM\     VarUI4FromStr          $6537170D
  ADVAPI32.dll  $BFED0000   C:\WINDOWS\SYSTEM\     VarUI4FromUI1          $65371598
  GDI32.dll     $BFF30000   C:\WINDOWS\SYSTEM\     VarUI4FromUI2          $65371704
  USER32.dll    $BFF60000   C:\WINDOWS\SYSTEM\     VarUdateFromDate       $6535C0A3
  KERNEL32.dll  $BFF70000   C:\WINDOWS\SYSTEM\     VariantChangeType      $6534F3AA
                                                   VariantChangeTypeEx    $653430F3
                                                   VariantClear           $653413C0
                                                   VariantCopy            $6534189E
                                                   VariantCopyInd         $65344A7F
                                                   VariantInit            $65343E78
                                                   VariantTimeToDosDat... $6536EFE3
                                                   VariantTimeToSystem... $6536F892
                                                   VectorFromBstr         $65371914
```

as constants (the compiler does special stuff with a lot of the contents of the System unit). As Listing 4 shows, we can overcome this problem by taking the address of each Variant and then de-referencing it. An irritating kludge, but never mind, it does the job. To unload the OLEAUT32.DLL, we pass its module handle (as returned from GetModuleHandle) to FreeLibrary and in doing so, it will hopefully take OLE32.DLL with it.

Figure 4 shows the Delphi 3 application running under Windows 9x with just 96Kb of committed memory (92Kb before any menus are accessed), and also shows Delphi 3's module list. Figure 3 showed Delphi 4's module list.

## Is That It?

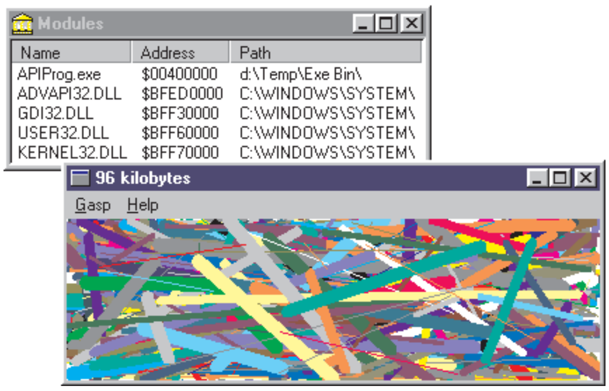
All this business of unloading the OLE DLLs works perfectly fine under Windows 95/98. Unfortunately, though, NT throws a spanner in the works. It seems that it is impossible to remove OLEAUT32.DLL from your address space, no matter how many times you pass its module handle to FreeLibrary.

Obviously this is a problem. Not an entirely insurmountable one, though. We can tackle Windows NT from another angle. In Issue 39, Hallvard Vassbotn's article *Slimming The Fat Off Your Apps* discussed the NT process working set. We can keep our memory usage down to an acceptable level by regularly reducing the process working set size. In fact, you can set the working set down to a value of zero, which means the process is swapped out of physical RAM. Any further code that needs to execute will then be pulled back in, as it is needed.

The point being made here is that any excess space taken up by little-used DLLs will be removed from RAM. Since the OLEAUT32.DLL code is only used as the program closes, its code and data should be left out of memory until the program is closing down. If OLEAUT32.DLL is the only cause of the RAM bloat, then setting the working set to 0 when the program starts up will do the required job.

So with this information to hand, let's get back to the code and see what needs to change. Normally, to detect if the program is running under Windows NT, you can use the Win32Platform variable from SysUtils. However, adding SysUtils to the uses clause for just one variable is excessive, it will pull in quite a lot of unnecessary code. So instead we can use the corresponding APIs to determine which platform we are on.

Listing 5 shows the modified version of ReduceMemoryOverhead (originally from Listing 4) with the extra NT support code in place. Remember that the application claimed to take up 4,684Kb of committed memory? Well, even with this new code in place it still does. However, the NT Task Manager shows that the program (as far as *it* is concerned) has dropped its memory size from 1,148Kb down to 464Kb. So, a definitely noticeable drop.



| Name | Address | Path |
|---|---|---|
| APIProg.exe | $00400000 | d:\Temp\Exe Bin\ |
| ADVAPI32.DLL | $BFED0000 | C:\WINDOWS\SYSTEM\ |
| GDI32.DLL | $BFF30000 | C:\WINDOWS\SYSTEM\ |
| USER32.DLL | $BFF60000 | C:\WINDOWS\SYSTEM\ |
| KERNEL32.DLL | $BFF70000 | C:\WINDOWS\SYSTEM\ |

➤ *Figure 4*

## And Finally...

In summary, if you are very, very careful, you can get applications written in Delphi 3 and later to consume an acceptably low amount of Windows memory. But it does take work. If Delphi's linker implicitly supported delay loading of DLLs, as does Visual C++, and as Hallvard Vassbotn wrote about in Issue 43 (*Delayloading Of DLLs*), the problem would not arise in the first place. OLEAUT32.DLL is only called as the program closes, and so would only be loaded, briefly, as the program shuts down.

## Acknowledgements

Thanks are due to Andy Strong for input and advice on this subject.

Brian Long is an independent consultant and trainer. You can reach him at brian@blong.com

➤ *Listing 5*

```
procedure ReduceMemoryOverhead;
var
  ProcessHandle: THandle;
  OSVersionInfo: TOSVersionInfo;
begin
  //Stop the RTL wanting to clear System Variants
 {$ifndef DelphiLessThan4}
  TVarData(EmptyParam).VType := varEmpty;
 {$endif}
  //These two seem to be considered constants, so we hack around
  //this by de-referencing the "constant" item's address
  TVarData((@Null)^).VType := varEmpty;
  TVarData((@Unassigned)^).VType := varEmpty;
  //Unload OLEAUT32.DLL, which will in turn unload OLE32.DLL
  FreeLibrary(GetModuleHandle('OLEAUT32.DLL'));
  OSVersionInfo.dwOSVersionInfoSize := SizeOf(OSVersionInfo);
  if GetVersionEx(OSVersionInfo) and
    (OSVersionInfo.dwPlatformID = VER_PLATFORM_WIN32_NT) then begin
    ProcessHandle := OpenProcess(PROCESS_ALL_ACCESS, False, GetCurrentProcessID);
    //Remove any unimportant code/data from memory
    SetProcessWorkingSetSize(ProcessHandle, -1, -1);
    CloseHandle(ProcessHandle);
  end
end;
```