Managing Your Memory

by David Selwood

One of the problems with both Delphi 1 and Delphi 2 is that a compiled application only has one data segment. This needn't be a problem with smaller applications, but on larger applications this restriction should be planned for from the initial stages of development. This article discusses application and Windows memory and how to minimise use of the data segment.

Memory Model

The memory model of a Delphi application will consist of one or more code segments and one data segment. Table 1 shows the data segment contents.

Segment	Description
Data	Area of memory used to store global variables (variables not declared within procedures or functions or classes) and typed constants. Note that sometimes this area is referred to as static data. Don't confuse this term with static variables as used in C.
Stack	The Stack is used to store local variables (variables declared within procedures or functions or methods) and is used for the temporary storage of return addresses for subroutines.
Heap	This is the local heap and is used by Windows controls and API routines beginning with `Local`. The local heap can grow dynamically if not large enough.

➤ *Table 1: Data segment contents*

The data segment hence contains three segments. It also contains a task header which is about 16 bytes in size that is used by Windows. In Delphi 1 the size of the data segment is limited to 64Kb and with Delphi 2 the limit is 1Mb. When the barrier is reached the error *Data Segment too large* is displayed and you will be forced to re-engineer your design. The difference between 64Kb and 1Mb is extensive, but Delphi 2 developers will hit this limit if large data structures are not implemented correctly.

Delphi can provide you with some memory usage details if you compile your program and then select `Options | Compile | Information`. Both the stack size and local heap size can be modified under `Options | Project` and `Options | Linker`. These can also be set with the `$M` compiler directive. The setting of the stack size and local heap size has no effect when compiling DLLs since these use the stack and heap of the calling application.

Note that there are two heaps, the Windows heap (often referred to as the global heap) and the local heap, sometimes called the application's heap. The global heap consists of extended and virtual memory and Delphi uses a segment sub-allocator algorithm for allocation of this memory. This is required because Windows imposes a system-wide limit of 8192 memory blocks. The routines `MemAvail` and `MaxAvail` return the amount of free memory available on the global heap. If you read this topic up, the documentation may refer to the heap, but might give no mention if it is the Windows or application's heap. In general you can assume they are referring to the Windows heap.

Minimising Data Segment Use

Setting the stack size is a hit and miss adventure. Tools are available that inform you how much stack your application uses, but these results tend to be inaccurate. When testing a project always have stack overflow checking turned on. This can be found under `Options | Project | Compiler` under `Run-Time Errors`.

I would suggest you keep making the stack smaller and test your application fully until the runtime error 202 occurs or you get a general protection error, which means a stack overflow error has occurred. You can then increase the size of the stack by 10%.

Again, setting the local heap size can be difficult. Borland recommend a minimum local heap size of 1024 bytes, if your application uses Windows edit controls or list boxes. Since this is generally true I would recommend that you don't allow the local heap to be declared smaller than this. Also you should have I/O error checking turned on. Should you use the Windows API functions beginning with `Local` (for example `LocalAlloc`) then the size should be increased accordingly. If the heap size is too small a general protection error will occur or a runtime error 203.

The static data size will depend on the global variables and typed constants in the Pascal units you have created and also the units you are accessing with the `uses` clause. It should be possible to limit the static data your application uses by using Windows global memory instead. However you may have no control over memory use in units declared in the `Uses` clause.

To resolve the limitation imposed by one data segment, memory can be taken from the Windows global heap. The limitation will now be the size of the virtual memory. Memory that is allocated dynamically with `New` and `GetMem` use the Windows global heap. When objects are created these are stored in the global heap, with the exception of the variable which points to the class. This variable will only use four bytes in the data segment.

Some other recommendations to reduce the size of the data segment are as follows:

- Allocate large data structures dynamically so that the Windows global heap is used.
- Minimise your use of typed constants.
- Minimise your use of global variables.
- When using variables of type `string` always declare the maximum size of the string, otherwise 255 bytes will be allocated.
- Use `PChars` where possible as these use the Windows global heap.
- Use class objects as these use the Windows global heap.
- If you use variables declare them in a class.
- Should your application use many global variables then create a class to hold them.

A short example of using the Windows global heap is given in Listing 1.

This only uses four bytes in the application's data segment, for the storage of a pointer. The `New` command causes memory to be allocated dynamically at runtime from the Windows global heap. Any memory that is allocated should be disposed of and as can be seen in the example the `Dispose` command is within a `Finally` block so that the memory will always be freed. If memory is not disposed of then memory leaks will occur and you will not be popular. To access the data you only have to use the caret operator (^) to deference the pointer.

Windows Task Database

One of the riddles with Windows is that it uses DOS conventional memory to store the Windows Task Database. Each Windows program (EXE or DLL) creates a task database and generally when Windows has booted up there can be as little as 150Kb free, so this valuable space should be monitored.

It is common knowledge that there are Windows API functions which return the percentage of free resources (`GetFreeSystemResources`) and the amount of global heap free (`GetFreeSpace`). However, there is no function which returns the amount of free conventional memory. The function in Listing 2 will calculate the amount of free base memory and should be used if your application relies on DLLs, OLE, or if it spawns off other applications.

Conclusion

Hopefully this article will have provided a brief insight into an application's data segment and ideas on how to overcome its shortcomings.

If possible experiment with some of the Windows memory monitoring tools such as InfoSpy and HeapTrace. Norton also provide a tool called SysWatch which monitors DOS memory. Knowing what memory is being used and when will allow you to avoid memory problems.

David Selwood currently works as a freelance contractor. Email him at dselwood@aol.com

➤ *Listing 1*

```
procedure Display_Information;
type
  tAddress = record
    Forename:  string [30];
    Surname:   string [30];
    Email:     string [30];
  end;
var
  pAddress: ^tAddress;
begin
  New (pAddress);
  try
    with pAddress^ do begin
      Forename:= 'David';
      Surname:= 'Selwood';
      Email:=   'DSELWOOD@AOL.COM';
      ShowMessage(Forename + ' ' + Surname + #13 + Email);
    end;
 finally
   Dispose(pAddress);
 end;
end; {Display_Information}
```

➤ *Listing 2*

```
Function GetBaseMemFree: longint;
var
  AllocatedBlock, Maximum, Minimum : longint;
  FreeBlock : word;
begin
  {Maximum amount of DOS memory = 1Mb}
  Maximum:= $100000
  Minimum:= $0;
  while Maximum > Minimum do begin
    Result:= (Maximum + Minimum) div 2;
    AllocatedBlock:= GlobalDosAlloc (Result);
    {Can't allocated block}
    if AllocatedBlock = 0 then
      Maximum:= Result
    else begin
      FreeBlock:= AllocatedBlock and $FFFF;
      GlobalDosFree (FreeBlock);
      Minimum:= Result +1;
    end;
  end; {while}
end; {GetBaseMemFree}
```