Startup And Shutdown Points

by Brian Long

There are always requirements for initialising things, for example calling the Randomize routine or creating a temporary file with a lot of commonly used data in. Similarly, there is always a need to do certain housekeeping tasks, tidying things up, like the aforementioned temporary file. Delphi has several entry and exit points that you can take advantage of in executables, and more in DLLs, although not all are documented. This article intends to cover all the standard Delphi compiler and RTL supported options, but ignores the object-based options such as OnCreate, OnActivate and OnShow events. We will start with the startup options.

Initialisation Parts

Delphi Pascal units have always had optional initialisation parts, in addition to their mandatory interface and implementation parts. If a unit has an initialisation part, and is added to the uses clause of any Pascal source file (unit or project source file) in a project, the statements in it will be executed when the program starts. Unit initialisation parts are executed during the begin of the begin...end block of the project source file, and this happens regardless of whether anything in the unit's interface part is referenced or not. Several VCL units have initialisation sections, as does the SysUtils run-time library unit.

The initialisation part appears at the end of the implementation part and is marked either by the word begin, which dates back to Turbo Pascal, or initialization, introduced in Delphi 1. The online help tells us that initialization is preferred, as its intent is clearer, so it appears a little odd that Borland use the old begin approach in many of the VCL units they supply the source for in Delphi. The main reason to use the new reserved word is to avoid Delphi messing up your form units. If you make a new project and add an initialisation part in the form unit using begin, Delphi is unable to add event handlers correctly. Listing 1 has an example unit with an initialisation section.

Double-clicking the form should make an OnCreate handler. We get the unfortunate mess shown in Listing 2. Using initialization avoids this and gives the rather better organised code in Listing 3.

You can put as many statements in the initialisation section as you like, without requiring an extra begin..end block.

InitProc And TApplication.Initialize

An alternative entry point, which occurs after the begin of the project source file (remember unit initialisation parts occur *during* the begin) but before any forms get created is through InitProc. This is a System unit pointer added in Delphi 2 that is designed to allow VCL objects to set themselves up, safe in the knowledge that all the VCL unit initialisation parts will have finished executing (as will initialisation parts of all other units). The OLE Automation server and COM server code in Delphi 2 and 3 uses this hook to parse any important command-line parameters, such as /REGSERVER or /UNREGSERVER.

The idea is to set InitProc up to point at your VCL setup routine in a unit initialisation part (saving the old value), and the routine will be called before any forms are created. The InitProc chain is invoked by a TApplication method called Initialize. A call to Application.Initialize is added to every new project in Delphi 2 and 3 as the first instruction in the project

Listing 1

```
unit Unit1;
interface
...
implementation
...
begin { start of initialisation section }
  Randomize; { initialisation code }
end. { end of unit }
```

Listing 2

```
unit Unit1;
interface
...
implementation
...
begin { start of initialisation section }
  Randomize; { initialisation code }
procedure TForm1.FormCreate(Sender: TObject);
begin
end;
end. { end of unit }
```

Listing 3

```
unit Unit1;
interface
...
implementation
...
procedure TForm1.FormCreate(Sender: TObject);
begin
end;
initialization { start of initialisation section }
  Randomize; { initialisation code }
end. { end of unit }
```

source file. Once you know this you can remove the line if you have no need for InitProc, and you are not writing an OLE or COM server application.

To set up InitProc you need a pointer variable to store the old value and a parameterless procedure whose address can be assigned to it. The procedure will need to call the old routine that used to be in InitProc (if any) using type TProcedure as a typecast, and then do whatever initialisation is required. Some code from the implementation section of a unit that does this is given in Listing 4.

ExitProc

Things are similar for exit hooks, but not identical. You might expect ExitProc to be the exact opposite of InitProc, operating in the same way, but you would be wrong. The System unit pointer ExitProc has been around for some time (ie it pre-dates Delphi 1), and is a bit more automatic than InitProc, which needs to be kicked off with a call to a TApplication method. ExitProc hooks are invoked after the project source file's code has finished executing, during the System unit shutdown procedure.

To set up an ExitProc hook, you need a pointer variable. At some point in the program's lifetime (and in fact usually in a unit initialisation part) you need to assign the address of your exit procedure to ExitProc after saving the old value in your pointer variable. If you are using Delphi 1, the exit procedure needs to be compiled in the far call model. The easiest way of ensuring this is the case is to put the far directive at the end of the procedure header. Note that far is ignored in Delphi 2 and 3. However, you can also declare the routine in the unit's interface section to achieve the same effect.

In the exit procedure, ExitProc must be assigned its old value before your shutdown code is performed. See Listing 5.

Notice that with ExitProc you don't call the old routine, unlike with InitProc (but you must set ExitProc to refer to the old routine). Inside the Delphi RTL is some assembler code to do it, which executes just before the program exits. You can consider it to be doing this:

```
while Assigned(ExitProc) do
  TProcedure(ExitProc);
```

Note that ExitProc should only be used in fixed EXE or DLL units: it is not compatible with Delphi 3 packages. So, in Delphi 3, ExitProc should only be used in a unit that has been marked with the $DenyPackageUnit compiler directive to avoid any catastrophes (Access Violations in fact). Packages must use AddTerminateProc or finalisation sections instead (see later).

AddExitProc

To make exit procedures a bit easier to add in, Delphi 1 introduced the AddExitProc routine to the SysUtils unit. AddExitProc adds an exit procedure to a chain of routines in a similar way to ExitProc, but maintains the list itself. As a result, your exit procedure need do nothing more than the tidying up code that you want to write. Your exit procedure, as with ExitProc, must be a parameterless routine and if in Delphi 1 must be compiled in the far call model. See Listing 6.

Routines added with AddExitProc will execute after the project source file has finished running but before procedures added through ExitProc. Note that the exit routines added are executed in reverse order: the last one added is executed first.

Note that AddExitProc should only be used in fixed EXE or DLL units: it is not compatible with packages. So, as before, in Delphi 3, AddExitProc should only be used in units marked with the $DenyPackageUnit compiler directive. Packages must use AddTerminateProc or finalisation sections instead.

AddTerminateProc

Delphi 3 adds this into the programmer's arsenal. You can use AddTerminateProc to add a function that returns a Boolean into the list of termination routines. The general purpose of these routines is to provide a generalised way of allowing or preventing your program from terminating. When your program is about to close, the VCL calls all the functions set up with AddTerminateProc to see if it is okay to exit. If any of them return False, the termination is stopped. This is even true if the termination was due to the user trying to close Windows. If any of the routines return False, Windows will not close.

The routines are called in the implementation of Application.Terminate and also in the wm_QueryEndSession message handler in the TCustomForm class. That means they execute before all the unit exit procedures and finalisation sections.

I suppose this is a more global way of providing an OnCloseQuery

➤ *Listing 4*

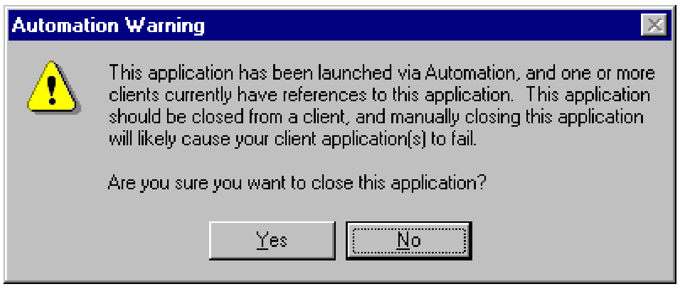
```
unit Unit1;
interface
...
implementation
...
var OldInitProc: Pointer;
procedure NewInitProc;
begin
  if Assigned(OldInitProc) then
    TProcedure(OldInitProc);
  { Do initialisation code here };
end;
initialization
  OldInitProc := InitProc;
  InitProc := @NewInitProc;
end.
```

➤ *Listing 5*

```
unit Unit1;
interface
...
implementation
...
var OldExitProc: Pointer;
procedure NewExitProc;
begin
  ExitProc := OldExitProc;
  { Do tidy-up code here };
end;
initialization
  OldExitProc := ExitProc;
  ExitProc := @NewExitProc;
end.
```

➤ *Listing 6*

```
unit Unit1;
interface
...
implementation
uses SysUtils;
...
var OldExitProc: Pointer;
procedure NewExitProc; far;
begin
  { Do tidy-up code here };
end;
initialization
  AddExitProc(NewExitProc);
end.
```


Automation Warning

This application has been launched via Automation, and one or more clients currently have references to this application. This application should be closed from a client, and manually closing this application will likely cause your client application(s) to fail.

Are you sure you want to close this application?

➤ *Figure 1*

Parameter Value	Reason for Parameter
dll_Process_Attach	DLL is being loaded explicitly by a call to `LoadLibrary`, or implicitly due to a process starting up.
dll_Process_Detach	DLL is being freed due to a process exit or a call to `FreeLibrary`.
dll_Thread_Attach	A thread (other than the main process thread) has been created in a process attached to the DLL.
dll_Thread_Detach	A thread is exiting cleanly in a process attached to the DLL.

➤ *Table 1: DllEntryPoint parameter values*

```
library Dll;
uses Windows;
procedure DllEntryPoint(Reason: DWord);
begin
  case Reason of
    dll_Process_Attach: { do something };
    dll_Thread_Attach:  { do something };
    dll_Thread_Detach:  { do something };
    dll_Process_Detach: { do something };
  end;
end;
begin
  if IsLibrary then begin
    // Set up the DLLEntryPoint routine
    DLLProc := @DLLEntryPoint;
    // Call it for process attachment (it won't happen automatically)
    DllEntryPoint(dll_Process_Attach)
  end
end.
```

➤ *Listing 9*

```
unit Unit1;
interface
...
implementation
uses SysUtils;
...
function NewTerminateProc: Boolean;
begin
  { Allow termination to proceed }
  Result := True;
  { Do tidy-up code here };
  end;
initialization
 AddTerminateProc(NewTerminateProc);
end.
```

➤ *Listing 7*

```
unit Unit1;
interface
...
implementation
...
initialization
{ startup code}
finalization
{ shutdown code }
end.
```

➤ *Listing 8*

handler for the main form, obviously without requiring any changes to the main form unit. The Delphi 3 `ComServer` unit uses it to bring up a warning if anyone tries to close a COM server app manually, rather than via the client app (see Figure 1). If the user is happy to close the server manually, the routine returns `True`. If they change their mind it returns `False`.

Given that the supplied Borland code only implements a termination handler for COM servers, you can use this approach as a general shutdown mechanism (Listing 7).

Finalisation Part

Delphi 2 and 3 have an optional unit part to complement the initialisation part. The `finalization` keyword signifies the beginning of a finalisation part. Finalisation parts must be placed after initialisation parts, and can only be added to a unit if there is an initialisation part. The obvious workaround if you want shutdown code but no startup code is to have an empty initialisation part.

Finalisation parts execute after the project source code, any `AddExitProc` routines and `ExitProc` routines have executed. Also, finalisation parts execute in the opposite order to initialisation parts. Listing 8 shows a finalisation part.

DLLs

In 16-bit Windows, C programmers have `LibMain` and `WEP` for their DLL startup and shutdown code. Apart from all the unit entry and exit hooks discussed above, Delphi 1's equivalent to `LibMain` is the project source file main block (the `begin..end` section). In Win32, C programmers have a routine called `DllEntryPoint` (also known as `DllMain`) which acts as both an entry and exit point for DLLs.

`DllEntryPoint` gets called under four circumstances, and is passed a parameter to indicate the circumstance (see Table 1).

Delphi does not have a direct `DllEntryPoint` equivalent, but we can make one using the 32-bit `System` unit pointer `DLLProc`. This pointer starts off as `nil`, but can be assigned the address of a routine taking a double word parameter. This routine effectively becomes `DllEntryPoint`, being called when necessary, except for one small problem.

`DllEntryPoint` should be called when a DLL gets loaded, but at that time `DLLProc` is `nil`. So our `DllEntryPoint` replacement will only be invoked for three out of the four circumstances. We can fudge our way around the problem by explicitly calling the routine after setting up `DLLProc` as shown in Listing 9.

Summary

Delphi provides a number of entry and exit hooks for programmers to take advantage of. To show them all in action, there are some example projects on this month's disk. STUBEXE32.DPR and STUBDLL32.DPR demonstrate all the hooks available for 32-bit DLLs and EXEs. Compile them both and then run STUBEXE32.EXE. This is a console mode application which writes out a line at each point of significance. Remember that since ExitProc and AddExitProc are both used, you must compile without run-time package support in Delphi 3, otherwise you will get Access Violations on program termination.

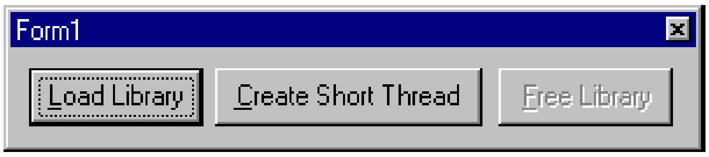
Figure 3 shows the output after each button in STUBEXE32.EXE (Figure 2) was pressed in turn and the application was closed.

STUBEXE.DPR and STUBDLL.DPR are 16-bit versions of the program and DLL. Because 16-bit applications can't use console mode, and because DLLs can't use the WinCrt unit, I made these files generate Windows debug strings at the relevant places. A debugging tool such as DBWin can capture these as shown in Figure 4. Note also that the program is a bit simpler since Win16 has no thread support and does not have finalisation sections or AddTerminateProc support.

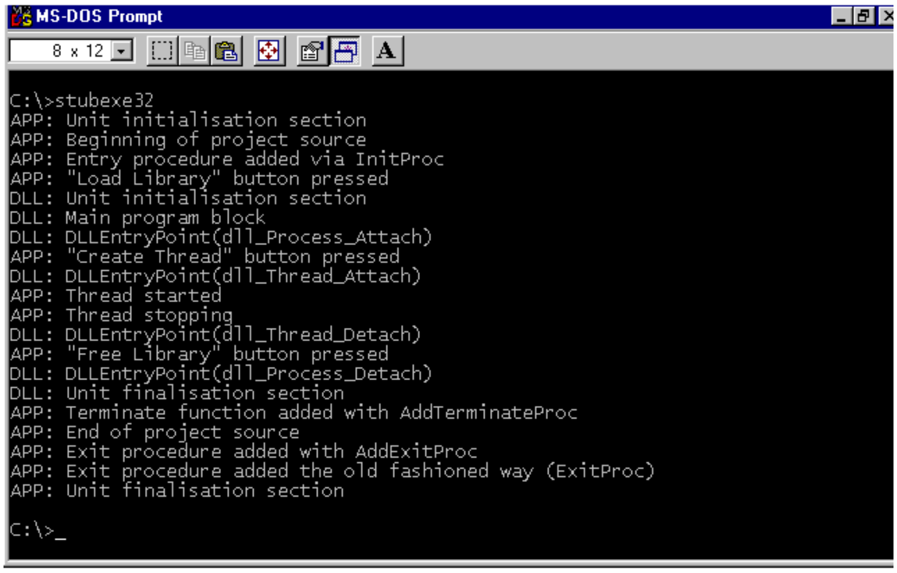
Acting When The Form Has Finished Drawing

Before we leave this article to rest, one last thing. There are various hooks into a form that trigger at various points in its lifetime. OnCreate is called in its constructor, OnShow is called when it has been asked to become visible, but hasn't actually got there. You can also override the Loaded method to act after all properties of all components on the form have been read in and set. But there is no event/method that lets you do some processing after the form has set itself up and finished drawing itself in its initial state. What do we do if we want such an event?

Well, you could make a wm_Paint message handler or OnPaint event handler, but that would be triggered every time the form needed painting. To achieve this possible requirement you can post yourself a custom message. The problem with OnShow is that it gets executed



➤ Figure 2

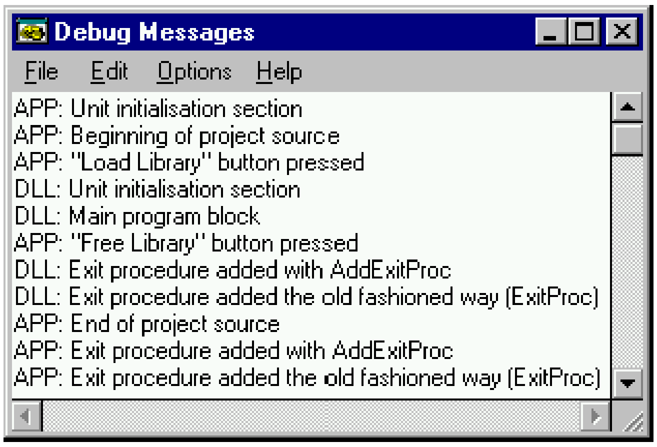


```
C:\>stubexe32
APP: Unit initialisation section
APP: Beginning of project source
APP: Entry procedure added via InitProc
APP: "Load Library" button pressed
DLL: Unit initialisation section
DLL: Main program block
DLL: DLLEntryPoint(dll_Process_Attach)
APP: "Create Thread" button pressed
DLL: DLLEntryPoint(dll_Thread_Attach)
APP: Thread started
APP: Thread stopping
DLL: DLLEntryPoint(dll_Thread_Detach)
APP: "Free Library" button pressed
DLL: DLLEntryPoint(dll_Process_Detach)
DLL: Unit finalisation section
APP: Terminate function added with AddTerminateProc
APP: End of project source
APP: Exit procedure added with AddExitProc
APP: Exit procedure added the old fashioned way (ExitProc)
APP: Unit finalisation section

C:\>_
```

➤ Figure 3

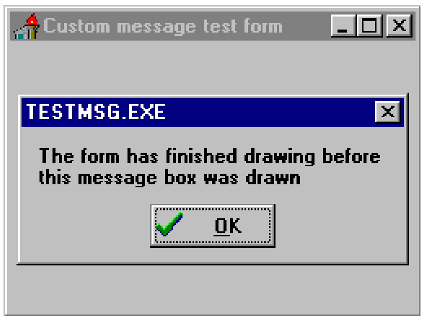
➤ Figure 4



```
APP: Unit initialisation section
APP: Beginning of project source
APP: "Load Library" button pressed
DLL: Unit initialisation section
DLL: Main program block
APP: "Free Library" button pressed
DLL: Exit procedure added with AddExitProc
DLL: Exit procedure added the old fashioned way (ExitProc)
APP: End of project source
APP: Exit procedure added with AddExitProc
APP: Exit procedure added the old fashioned way (ExitProc)
```

```
unit TestMsgU;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, StdCtrls;
const
  wm_CustomMsg = wm_User + $999;
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    procedure WMCustomMsg(var Msg: TMessage); message wm_CustomMsg;
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  PostMessage(Handle, wm_CustomMsg, 0, 0);
end;
procedure TForm1.WMCustomMsg(var Msg: TMessage);
begin
  ShowMessage(
    'The form has finished drawing before '#13'this message box was drawn');
end;
end.
```

➤ Listing 10

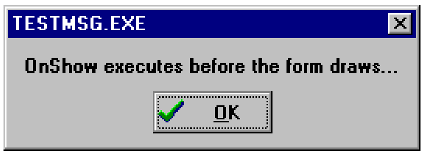

Custom message test form

TESTMSG.EXE

The form has finished drawing before this message box was drawn

➤ *Above: Figure 5*

➤ *Below: Figure 6*


TESTMSG.EXE

OnShow executes before the form draws...

immediately after the form is told to paint itself. This would be okay, but the form is told to paint in an indirect way that causes a `wm_Paint` message to be posted into the form's message queue and therefore not processed straight away. So the `OnShow` handler executes and then at some later point the `wm_Paint` message gets processed and the form draws. What we can do is to cause a message of our own to also be posted into the form's message queue, but ensure it gets in after the `wm_Paint`. Posting a message from the form's `OnCreate` event handler seems to do as we require, which only leaves writing an appropriate message handler to do the rest.

An example form unit from TESTMSG.DPR is shown in Listing 10. It has a message handler that simply brings up a modal message box. If you run the program, you will find the message box appears immediately after the form has first drawn itself (see Figure 5). If you try putting a call to `ShowMessage` in an `OnShow` handler, it will appear before the form draws itself (see Figure 6).

Brian Long is a UK-based freelance Delphi and C++ Builder consultant and trainer. He is available for bookings and can be contacted at brian@blong.com. Professional enquiries can go to consultancy@ blong.com or training@blong .com