Under Construction: Delphi Wizard Magic

by Bob Swart

This month I'm going to explain the difference between a plain Expert and a real Wizard. We'll develop a Wizard Form/Component Template and end up by seeing how to write a single Wizard DLL Expert that can be installed into both Borland C++ Builder and Delphi – and I don't mean single source code, but single binary code compatible!

Borland tools always used to have Experts (such as the Target Expert and Class Expert in Borland C++, and the Component Expert and Table Form Expert in Delphi), while Symantec called them Assistants and Microsoft named them Wizards. Alas, Borland C++ Builder and now Delphi 3 have introduced us to the new official Borland terminology: Wizards. Like it or not, Microsoft does set the naming conventions. Anyway, Delphi has a `TIExpert` interface (no need to change the class name as well and break existing code) that we can use to write Experts, ahem, Wizards that is. We've seen the techniques already in Issues 3, 7, 11 and 13, but let's now explore the true nature of Wizards and see how we can build them ourselves.

Wizard GUI

Having a `TIExpert` dialog pop up in the Delphi IDE is certainly nice, but doesn't constitute a real Wizard. If we look at the Microsoft Windows 95 *Add New Hardware* Wizard (Figures 1 and 2) we can identify some issues and attributes that are important when designing a consistent Wizard interface.

First of all, the Wizard is a form with a fixed size, comprising two major parts: the bottom part with the navigation buttons (`Back`, `Next`, `Cancel`, sometimes `Help`, and `Finish` at the end) and the upper part with a bitmap on the left and the actual Wizard contents on the right.

➤ *Figure 1*

➤ *Figure 2*

Button	Action
< Back	Returns to the previous page (remove or disable button on the first page)
Next >	Moves to next page in the sequence, maintaining whatever settings the user provide in previous pages
Finish	Applies user-supplied or default settings from all pages and completes the task
Cancel	Discards any user-supplied settings, terminate the process and closes the wizard window

➤ *Table 1: Wizard button actions*

Looking closer we notice that the bitmap which is displayed on the left can be specific to the entire Wizard or just to the current step. The remainder of the form guides the user. So a Wizard is not just another data entry form, but a form where we first give an explanation of what kind of input we expect, followed by the input control to enter this value (for example a file to be copied). This way we can see the Wizard as an "Expert" that can help and guide inexperienced users (who may need to read the cues), while experienced users can just fill in the input controls and go to the next page. All in all, a Wizard is a well-established user interface design element, open to all kinds of uses.

MS Wizards

The *Windows Interface Guidelines for Software Design*, published by Microsoft Press (ISBN 1-55615-679-0), has a lot of additional useful information. According to this book a Wizard is a series of pages which help the user through a task. The pages include controls to gather input from the user which is then used to complete the task. Table 1 shows the definitions of the standard command buttons which appear at the bottom of the Wizard form.

The book further advises us to use the title bar text of the Wizard form to identify the purpose of the Wizard and optionally the purpose of the specific step that is being performed on the current page.

On the first page of the Wizard, we should include a graphic in the left side of the form to establish a reference point. On the top right portion of the Wizard window a short paragraph welcomes the user to the Wizard and explains

what it does. We can also include controls for entering or editing initial input to be used by the Wizard, if there is sufficient space.

On subsequent pages, we can continue to include a graphic for consistency or, if space is critical, use the entire width of the form for displaying instructional text and controls for user input. When using graphics, we should make sure to include pictures that help illustrate the process. Also, we should include default values or settings for all controls where possible.

We can include the `Finish` button at any point that the Wizard can complete the task, placed to the right and adjacent to the `Next` button. This allows the user to step through the entire Wizard or only the page on which they wish to provide input. Otherwise, if the user needs to step through each page of the Wizard, we should replace the `Next` button with the `Finish` button on the last page of the Wizard. Also on the last page of the Wizard we must indicate to the user that the Wizard is prepared to complete the task and instruct the user to click the `Finish` button.

In summary, we must design the Wizard pages to be easy to understand. It is important that users immediately understand what a Wizard is about so they don't feel like they have to read it very carefully to understand what they have to answer. It is better to have a greater number of simple pages with few choices than a smaller number of complex pages with too many options or text.

TWizard

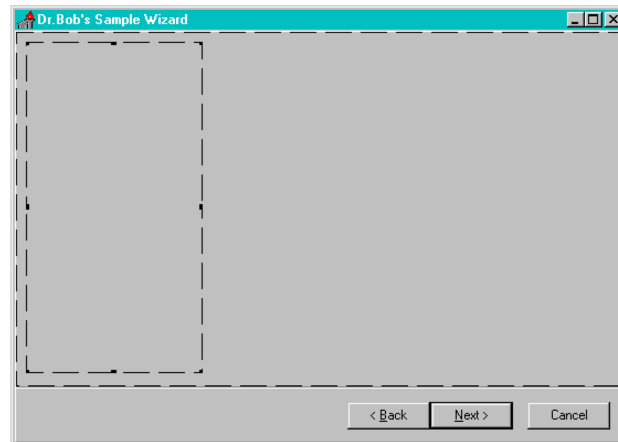
Now that we've learned how to design a Wizard according to the Microsoft guidelines, let's find out how much work it takes to actually create something that looks and acts like a Wizard. In order to do that, lets take a form, set `Width` to 480, `Height` to 360, `Scaled` to `False`, `Position` to `poScreenCenter`, `BorderStyle` to `bsDialog` and the `Caption` to `Dr.Bob's Sample Wizard`. Then, drop a `TBevel` on the form, set `Shape` to `bsTopLine` and `Align` to `alBottom`. Now, drop three `TButton` controls on the form just below the `Bevel` and set the captions (from left to right) to `< &Back`, `&Next >` and `Cancel`. Note that we have to set the form's `BorderStyle` to something other than `bsSizeable` or `bsSizeToolWin` (such as the `bsDialog`), otherwise we would not be able to drop anything below the `Bevel` line – try it and you'll see that the `Bevel` shrinks down to below the buttons. Why? I first suspected a bug in Delphi, but Danny Thorpe of Borland R&D told me that this behaviour is due to AutoScroll, which is enabed by default on resizeable form styles and disabled by default on fixed-size form styles.

Anyway, make sure the `Back` and `Next` buttons touch each other and the `Cancel` one is a little to the right. Ensure the `Next` button is the default button and set the `ModalResult` property of the `Cancel` button to `mrCancel` (so it cancels the Wizard when clicked). Now, put a `TImage` on the left part of the form just above the `Bevel` and you have a Wizard skeleton almost ready to use.

The final step involves going to the `Win 3.1` tab of the component palette and dropping a `NoteBook` on the form. Set the `Align` property to `alClient`. The `TImage` will suddenly disappear (behind the `NoteBook`)! We can right click on the `NoteBook` and choose `Send to Back` all we want, the `TImage` will not reappear. We need to put the `TImage` on a `Panel` first (ie remove the `TImage`, remove the `Notebook`, drop a `Panel` on, drop the `TImage` on the `Panel`, drop a `NoteBook` on, set `Align` to `alClient`, perform `Send to Back` and the `Panel` with the `TImage` will reappear). Now, we have a `Panel` with a `TImage` that is placed on the *form* and not on the `NoteBook`, so it will show regardless of which `NoteBook` page is the current `ActivePage`. This technique can be used only if we have a single image for the entire Wizard. If we want to use different images for different steps pages of the Wizard then we should just drop a `TImage` on every page of the `NoteBook` that needs one. The latter option requires more resources and generates a bigger executable. So, we now have our Wizard Skeleton, as shown in Figure 3.

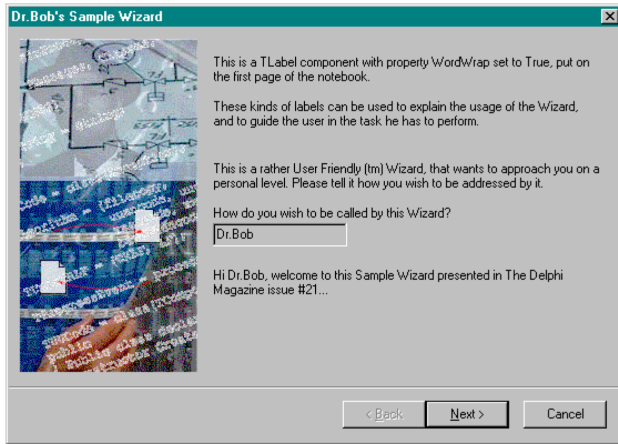
There's one important last step, which doesn't have anything to do with designing the Wizard but with the flow of control: the `Next` and `Back` buttons must share the same `ButtonStepClick` event handler. But

➤ Figure 3

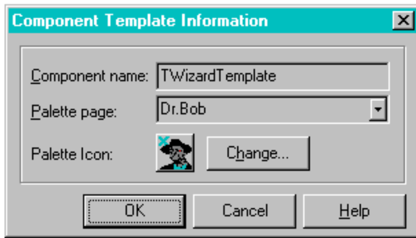



```
procedure TWizardForm.ButtonStepClick(Sender: TObject);
begin
  if Sender IS TButton then
    NoteBook.PageIndex := NoteBook.PageIndex + (Sender AS TButton).Tag
  else
    NoteBook.PageIndex := 0;
  ButtonBack.Enabled := NoteBook.PageIndex > 0; { first }
  if NoteBook.PageIndex < Pred(NoteBook.Pages.Count) then begin
    ButtonNext.Caption := '&Next >';
    ButtonNext.ModalResult := mrNone
  end else begin
    { Finish }
    ButtonNext.Caption := '&Finish';
    ButtonNext.ModalResult := mrOk
  end
end;
```

➤ Listing 1



➤ *Figure 4*



➤ *Figure 5*

```
program oz;
uses
  Controls, Dialogs, Forms,
  wizard in 'wizard.pas';
begin
  with TWizardForm.Create(nil) do
  try
    ShowModal
  finally
    Free
  end
end.
```

➤ *Listing 2*

first, give the `Tag` property of the `Next` button a value of 1, and the `Tag` property of the `Back` button a value of -1. If we click on the `Next` button, we want to advance one page and if we click on the `Back` button we need to step back one page (or advance -1 page: the value of the `Tag` property). The combined event handler only needs to cast the `Sender` to a `TButton`, check its `Tag` property and do the required processing based on that. If the `Sender` is not a `TButton`, then the `NoteBook` would be set to display the first page. This can be used for initialisation in the `FormShow` event, for example, where `ButtonStepClick` can be called with the form itself or with a `nil` argument. See Listing 1.

Note that the first `NoteBook` page has `PageIndex` 0, while the last page has `PageIndex` equal to `NoteBook.Pages.Count` minus one. If we're on the last page, the `Next` button changes into the `Finish` button – changing the caption to `Finish` and `ModalResult` to `mrOk`. Of course, we need to ensure that the `Finish` button changes back into the `Next` button the user clicks `Back`.

This one event handler is usually all the code you need! Sometimes, the `Next` button can only be clicked on after some action has been performed (like a pre-condition). You need to program that explicitly yourself. You can for example set the `Enabled` property of the `Next` button to `False` by default and only enable it when the user has fulfilled the requirements of the specific Wizard page, which you can check in the `OnChange` event handlers of the controls and components on that particular page.

The complete source code for the `TWizardForm`, which is nothing more than a Wizard template, is of course on this month's disk. An example project to show the Wizard in action is shown in Listing 2 (it compiles with all versions of Delphi and Borland C++ Builder). Given some more controls on the Wizard pages, we could end up with a sample Wizard like Figure 4.

Of course, version 3 of my collection of new Delphi and C++ Builder Wizards will be using a similar layout and Wizard style!

Delphi 3 Component Template

Delphi 3 has the new ability to create compound components (also called super components) on the fly. Just select the components you want to be included in the component template and select `Component | Create Component Template`. The result is a new pseudo-component including all code for the event handlers we've just written. A great new way to add useful combinations of components and properties to the component palette. Especially in this case, where we can save the entire Wizard form we've built so far as one `TWizardTemplate` component, just like that! See Figure 5.

Now let's try something new. Create a new form, but this time resize it to be larger than our initial Wizard Form of 480x360, say we make the height about 480 as well. Drop the new `TWizardTemplate` component on the form and notice that the buttons are suddenly children of the `Panel`. You cannot move the buttons back onto the `Bevel`, since the `Parent` of each button has become the `Panel`.

This is because when we drop the component template on the new (larger) form the components get created in this order: the `Bevel` and `Panel` *before* the buttons. But when the `Bevel` and `Panel` are adjusted according to their `Align` properties, the `Panel` is in the place where the `Bevel` was, on the spot where the buttons are about to be created. And indeed the buttons are created on top of the `Panel` instead of on top of the `Bevel`. The component template creator couldn't know that, it only reads the order, type and properties of the components that make up the component template.

The fix is to put everything (ie the entire previous `TWizardTemplate`) onto an extra `Panel` first: a `Panel` component that has its `Align` property set to `alNone`, so that the other template sub-components are created correctly. We can always set the `Align` property of the "fathering" component to `alClient`, if needed, after we've dropped it onto a new form.

PlugIn Wizard Expert

Well, now that we've written the Wizard interface form, we still need the expert interface to the Delphi (or C++ Builder) IDE. This is something we've done in the past, as I've noted. This time, we only need a

simple esStandard type expert that does nothing in its Execute method but call the same code that we used in the example project, except this time we want to connect the Wizard to the name of the currently loaded project. So, we call ToolServices.GetProjectName as well, which returns a string containing the name of the current project. See Listing 3.

Note the special extra exports entry of InitExpert as name 'INITEXPERT0017' which is needed for Delphi 3 since the internal version number of Experts/Wizards has changed from Delphi 2.01 and C++ Builder. Some other things have changed as well, but nothing that we need to fear at this time.

The Wizard form, example project and PlugIn expert DLL can be compiled with all versions of Delphi and C++ Builder. If we want to actually use the expert DLL then we need to add a line to DELPHI.INI (in the WINDOWS/SYSTEM directory):

```
WIZARD=C:\DELPHI\PLUGIN.DLL
```

where the path to PLUGIN.DLL must be correct, of course. For the 32-bit tools we need to add a Key to the registry at the Expert location for each of the tools. The Key name doesn't matter, but the value must be the exact path where the expert DLL resides.

Borland C++ Builder

Now that we have a 32-bit expert DLL for our Wizard, can we actually use it with C++ Builder and Delphi 3? The PlugIn Wizard expert compiles with C++ Builder with no problems and it runs fine as well. The sizes of the two DLLs (one compiled with Delphi 2.01 the other with C++ Builder 1.0) are different, however, and this seems to indicate that there is some difference in either the RTL/VCL or the compiler of Delphi 2.01 and C++ Builder.

Which leaves the question: are the expert DLLs created by the two compilers binary compatible? That is, can I use the PlugIn Wizard expert DLL compiled with Delphi 2.01 in C++ Builder, or vice versa? At first sight, this appears to be so (check out TRIPLEX.DLL by John Howe, which is one DLL that is binary compatible with Delphi 2, Delphi 3 and C++ Builder).

However, there are some compatibility problems that can arise, such as the use of ShareMem and esStandard experts (that get installed on the Help menu). We can prove this by trying to run the PLUGIN.DLL compiled with Delphi 2.01 in the C++ Builder IDE: we get access violations and *This Program Will Be Shut Down* messages all over the place. Not a pretty sight and a darn puzzle if you don't know where to look for the cause...

ShareMem

Every expert DLL which communicates with the Delphi or C++

➤ *Listing 3*

```
library plugin;
uses
  {$IFDEF WIN32}{$H+} ShareMem, {$ENDIF}
  Wizard, ExptIntf, ToolIntf, VirtIntf, Forms,
  Dialogs, SysUtils;
procedure HandleException;
begin
  if Assigned(ToolServices) then
    ToolServices.RaiseException(ReleaseException)
end {HandleException};
Type
  TPlugInExpert = class(TIExpert)
  public
    { Expert Style }
    function GetStyle: TExpertStyle; override;
    { Expert Strings }
    function GetIDString: String; override;
    function GetName: String; override;
  {$IFDEF WIN32}
    function GetAuthor: String; override;
  {$ENDIF}
    function GetMenuText: String; override;
    function GetState: TExpertState; override;
    procedure Execute; override; { Launch the Expert }
  end {TPlugInExpert};
function TPlugInExpert.GetStyle: TExpertStyle;
begin
  Result := esStandard
end {GetStyle};
function TPlugInExpert.GetIDString: String;
begin
  Result := 'DrBob.TPlugInExpert'
end {GetIDString};
function TPlugInExpert.GetName: String;
begin
  Result := 'Dr.Bob''s PlugIn Wizard Expert'
end {GetName};
{$IFDEF WIN32}
  function TPlugInExpert.GetAuthor: String;
  begin
    Result := 'Dr.Bob'
  end {GetAuthor};
{$ENDIF}
function TPlugInExpert.GetMenuText: String;
begin
  Result := 'No Project Wizard Available';
  if Assigned(ToolServices) then
    with ToolServices do
      try
        if (GetUnitCount > 0) and
          (GetFormCount > 0) and
          (Length(GetProjectName) > 0) then
          Result := '&Project Wizard for ' +
            ExtractFileName(GetProjectName) + '...'
      except
        HandleException
      end
end {GetMenuText};
function TPlugInExpert.GetState: TExpertState;
begin
  Result := [];
  if Assigned(ToolServices) then
    with ToolServices do
      try
        if (GetUnitCount > 0) and
          (GetFormCount > 0) and
          (Length(GetProjectName) > 0) then
          Result := [esEnabled]
      except
        HandleException
      end
end {GetState};
procedure TPlugInExpert.Execute;
begin
  if Assigned(ToolServices) then
    with ToolServices do
      try
        with TWizardForm.Create(nil) do
          try
            ShowModal
          finally
            Free
          end
      except
        HandleException
      end
end {Execute};
function InitExpert(Delphi: TIToolServices; RegisterProc:
  TExpertRegisterProc; var Terminate: TExpertTerminateProc):
  Boolean; {$IFDEF WIN32} stdcall; {$ELSE} export; {$ENDIF}
begin
  ExptIntf.ToolServices := Delphi; { Save! }
  if ToolServices <> nil then
    Application.Handle := ToolServices.GetParentHandle;
  Result := RegisterProc(TPlugInExpert.Create)
end {InitExpert};
exports
  InitExpert name ExpertEntryPoint resident,
  InitExpert name 'INITEXPERT0017'; { for Delphi 3 }
begin
end.
```

Builder IDE and wants to share (long) strings must include ShareMem as the first unit in its interface section, thereby making sure that the code inside the ShareMem unit shown in Listing 4 is the first to execute.

SharedMemoryManager contains three APIs (SysGetMem, SysFreeMem and SysReallocMem) that are implicitly loaded from the Delphi Memory Manager DLL that serves as a gateway between the Delphi IDE and the expert DLL. Very convenient, and if you ever write a DLL expert that causes a lot of access violations when talking to ToolServices, for example, then the presence or lack of the ShareMem unit should be the first thing to check.

ShareMem <> ShareMem

The compatibility problem between Delphi 2.x and C++ Builder is caused by the fact that their ShareMems are not the same. The Delphi 2.x version contains the import declarations for the DELPHIMM.DLL Delphi memory manager DLL, shown in Listing 5.

The C++ Builder version is somewhat different and is based on BCBMM.DLL not DELPHIMM.DLL, as shown in Listing 6. The first three APIs have different (mangled?) names and there's a new API called DumpBlocks.

So, if we compile an expert DLL that uses ShareMem with Delphi 2.x it probably won't run with C++ Builder and vice versa. Some things may work fine (esAddIn experts, for example) but our example doesn't!

New ShareMem

Since I would very much prefer a single binary copy of my expert DLL containing a collection of Wizards, I had to find a way to combine these two versions of ShareMem. Not loading the memory management DLL implicitly, like the original ShareMem, but loading it explicitly and first trying to find out which IDE is running: Delphi or C++ Builder.

Finding out which IDE is running can be approached in many different ways. One is to call the ToolServices API GetBaseRegistryKey and see if the string contains the substring Delphi or C++. We can even try to obtain the version number of the tool once we're inside the registry. Unfortunately, we can't call this API since it returns a long string, and we need to install the new memory manager before we can access long strings...

Another way is to do a FindWindow for a specific Window title, like Delphi 2.0 or C++Builder. Unfortunately, this technique won't work if both tools are running, so we have to skip that one too.

A final way I could think of was to check the value of ParamStr(0) from within the expert DLL. This string should hold the value of the calling application, just like the old DOS CommandLine string. This technique has just one flaw: ParamStr(0) returns... a long string, and we can't do anything with long strings before the new memory manager is installed (otherwise we'll get a lot of *run-time error 204 - invalid pointer operations* messages when we finally install the new memory manager, resulting in a fatal error for our expert DLL which will then terminate nicely). Fortunately, there's a nice API hidden in KERNEL32.DLL that returns the entire command line as a PChar string, one that we are allowed to handle before installing the memory manager: GetCommandLineA. All we need to do is find out whether or not the characters B, C and B occur next to each other in this string (since C++ Builder is BCB.EXE). This technique works, resulting in a new ShareMem unit that is compatible with both Delphi 2.x and C++ Builder (and probably Delphi 3, though I don't have the release version of Delphi 3 at the time of writing). See Listing 7.

Note that this unit also contains a Boolean variable Delphi in its interface section that you can use inside your expert DLLs to identify whether or not your Wizards should behave inside a Delphi or C++ Builder environment.

Results

We've seen what Wizards are, examined some guidelines when designing their user interfaces, and

➤ *Listing 4*

```
const
  SharedMemoryManager: TMemoryManager = (
    GetMem: SysGetMem;
    FreeMem: SysFreeMem;
    ReallocMem: SysReallocMem);
initialization
  SetMemoryManager(SharedMemoryManager);
end.
```

➤ *Listing 5*

```
const
  DelphiMM = 'delphimm.dll';
function SysGetMem(Size: Integer): Pointer; external DelphiMM;
function SysFreeMem(P: Pointer): Integer; external DelphiMM;
function SysReallocMem(P: Pointer; Size: Integer): Pointer; external DelphiMM;
function GetHeapStatus: THeapStatus; external DelphiMM;
function GetAllocMemCount: Integer; external DelphiMM;
function GetAllocMemSize: Integer; external DelphiMM;
```

➤ *Listing 6*

```
const
  DelphiMM = 'bcbmm.dll';
function SysGetMem(Size: Integer): Pointer; external DelphiMM
  name '@System@SysGetMem$qqri';
function SysFreeMem(P: Pointer): Integer; external DelphiMM
  name '@System@SysFreeMem$qqrpv';
function SysReallocMem(P: Pointer; Size: Integer): Pointer; external DelphiMM
  name '@System@SysReallocMem$qqrpvi';
function GetHeapStatus: THeapStatus; external DelphiMM;
function GetAllocMemCount: Integer; external DelphiMM;
function GetAllocMemSize: Integer; external DelphiMM;
procedure DumpBlocks; external DelphiMM;
```

how to implement them as Delphi and C++ Builder Wizard DLL experts. One expert DLL, multiple environments. Sounds like Magic, but that's what Wizards are for. And remember it takes a `TIExpert` to produce one...

Next Time, Dr.Bob Says...

Next time, we'll be looking into a fresh new Delphi 3 topic: how to turn our Wizard into an ActiveX, or better yet, into an ActiveForm, and then how to deploy it on the internet and/or an intranet.

Stay tuned!

Bob Swart (home.pi.net/~drbob/) is a professional knowledge engineer and technical consultant using Delphi and C++ Builder for Bolesian (www.bolesian.com), a freelance technical author and co-author of *The Revolutionary Guide to Delphi 2*. Bob is now co-working on *Delphi Internet Solutions*, a new book about Delphi and the internet/intranet. In his spare time, Bob likes to watch videos of *Star Trek Voyager* and *Deep Space Nine* with his 3 year old son Erik Mark Pascal and his 6 month old daughter Natasha Louise Delphine.

Threads Whoops...

Thanks very much to Luca Guzzon and Berend de Boer for noticing an oversight in last month's *Under Construction* column. In `TFirstThread.Create` in the source file THREAD1.PAS and also in the method `TSpagettiThread.Create` in the source file SPAGETTI.PAS I placed the

```
inherited Create(False);
```

statement at the beginning of the method.

This is unfortunately wrong: it has the effect of starting the thread immediately without the parameters being set. The solution is to move the

```
inherited Create(False);
```

statement to the last line of the two constructors.

Sorry folks!

➤ *Listing 7*

```
unit ShareMem;
{ (c) 1997 by Bob Swart (aka Dr.Bob, http://home.pi.net/~drbob/ }
interface
const
  Delphi: Boolean = True; { can be used outside the unit as well }
  SysGetMem: function(Size: Integer): Pointer = nil;
  SysFreeMem: function(P: Pointer): Integer = nil;
  SysReallocMem: function(P: Pointer; Size: Integer): Pointer = nil;
  GetHeapStatus: function: THeapStatus = nil;
  GetAllocMemCount: function: Integer = nil;
  GetAllocMemSize: function: Integer = nil;
  DumpBlocks: procedure = nil;
implementation
uses Windows;
const
  Handle: THandle = 0;
  SharedMemoryManager: TMemoryManager = (
    GetMem: nil;
    FreeMem: nil;
    ReallocMem: nil);
function GetCommandLine: PChar; stdcall;
  external 'kernel32.dll' name 'GetCommandLineA';
var P: PChar;
    i: Integer;
initialization
  P := GetCommandLine;
  i := 0;
  repeat
    Inc(i)
  until (P[i] = #0)
    or ((P[i] = 'B') and (P[i+1] = 'C') and (P[i+2] = 'B'));
  Delphi := P[i] = #0;
  if not Delphi then begin
    Handle := LoadLibrary('BCBMM.DLL');
    if Handle = 0 then
      MessageBox(HWnd(0),'Error: could not load BCBMM.DLL',
        nil,MB_OK or MB_ICONHAND);
    @DumpBlocks := GetProcAddress(Handle, 'DumpBlocks');
    @SysGetMem := GetProcAddress(Handle, '@System@SysGetMem$qqri');
    @SysFreeMem := GetProcAddress(Handle, '@System@SysFreeMem$qqrpv');
    @SysReallocMem := GetProcAddress(Handle, '@System@SysReallocMem$qqrpvi');
  end else begin
    { Delphi }
    Handle := LoadLibrary('DELPHIMM.DLL');
    if Handle = 0 then
      MessageBox(HWnd(0),'Error: could not load DELPHIMM.DLL',
        nil,MB_OK or MB_ICONHAND);
    @SysGetMem := GetProcAddress(Handle, 'SysGetMem');
    @SysFreeMem := GetProcAddress(Handle, 'SysFreeMem');
    @SysReallocMem := GetProcAddress(Handle, 'SysReallocMem');
  end;
  @GetHeapStatus := GetProcAddress(Handle, 'GetHeapStatus');
  @GetAllocMemCount := GetProcAddress(Handle, 'GetAllocMemCount');
  @GetAllocMemSize := GetProcAddress(Handle, 'GetAllocMemSize');
  SharedMemoryManager.GetMem := @SysGetMem;
  SharedMemoryManager.FreeMem := @SysFreeMem;
  SharedMemoryManager.ReallocMem := @SysReallocMem;
  SetMemoryManager(SharedMemoryManager);
finalization
  FreeLibrary(Handle)
end.
```