Tips & Tricks

TApplication's "Missing" Events

Delphi's help files are infamous for their missing links: here are a few more! The following events are all defined in FORMS.PAS in the VCL source and all have help file entries, but none of them are in the events list for the `TApplication` entry. Search for:

- `OnMinimize`: called when the app is minimised,
- `OnRestore`: called when the application is being restored after a minimisation,
- `OnShowHint`: allows you to control various hint window parameters, like how wide the hint window can get before word wrapping starts (see the help entry for `TShowHintEvent` for more details).

Delphi 2 adds several new methods and properties and, not surprisingly, the help file entries leave something to be desired here as well. The following properties are new to Delphi 2 and have links to their help topics from the `TApplication` topic:

```
property HintShortPause : Integer;
property HintHidePause : Integer;
property UpdateFormatSettings : Boolean;
```

These methods and property have help entries but the link from `TApplication` is broken:

```
procedure CreateHandle;
procedure Initialize;
property ShowMainForm : Boolean;
```

These methods have no help file entries and seem to have been incorrectly declared as `public`, they should probably be `private` methods:

```
procedure HideHint;
procedure HintMouseMessage(Control: TControl;
  var Message: TMessage);
```

This method has an incorrect declaration in the Delphi 2 help, it should be:

```
procedure CreateForm(InstanceClass:
  TComponentClass; var Reference);
```

I came across the "missing" events when I was trying to get application minimisation and restoration to work properly.

If you minimise an application from its main form or from a non-modal form (ie one displayed by calling the `TForm` method `Show`) everything works as expected. I needed to be able to minimise the application from a modal form (ie one displayed by calling the `TForm` method `ShowModal`). What happens is that the modal form gets minimised but the main form stays displayed and is unable to receive the focus. Now `TForm` does not have an `OnMinimise` event, but it does have an `OnResize` event which gets called when a form is minimised. I put the code in Listing 1 into the event handler on the modal form, and behold: minimisation works!

Restoring the application doesn't, though. The solution I found to this problem is to attach an event handler to the newly found `OnRestore` event of the `Application` variable (see Listing 2). This code uses the `Screen` variable to get a list of the application's visible forms and sends each one a message (via the form's `Perform` method) that tells it to restore itself.

Remember that you must attach event handlers to the `Application` variable in code, because it is not a visual component. In your application's main form, you should declare and define the event handler and attach it in the form's `OnCreate` event. Listing 2 shows how it's done. If you don't much like this method, there is source code on the disk for a component (file APPCOMP.PAS) that you can drop on your application's main form, so you can modify properties and attach event handlers at design time. The `Register` procedure controls where on your component palette `TAppComponent` will appear. To have it display on another tab, change `System` to the name of your preferred tab. Giving it a name that does not exist will create a new tab.

Contributed by Jim Cooper, Sybiz Software, CompuServe 101641,440

➤ *Listing 1*

```
procedure TMyModalForm.FormResize(Sender: TObject);
begin
  if WindowState = wsMinimized then
    Application.Minimize;
end;
```

➤ *Listing 2*

```
unit MainForm;
interface
type
  TMyMainForm = class(TForm)
    {... Various declarations}
public
  procedure AppOnRestore(Sender : TObject);
end;
implementation
procedure TMyMainForm.AppOnRestore(Sender : TObject);
var i : Integer;
begin
  {Loop through the all the forms in the application}
  for i := 0 to Screen.FormCount - 1 do begin
    if Screen.Forms[i].WindowState = wsMinimized then begin
      Screen.Forms[i].Perform(WM_SYSCOMMAND,SC_RESTORE,0);
    end;
  end;
end;
procedure TMyMainForm.FormCreate(Sender : TObject)
begin
  Application.OnRestore := AppOnRestore;
end;
end.
```

Heap Checking

I recently developed a unit to check the heap while debugging in Delphi 2 applications, eventually storing the results in a log file. The unit (MMANAGER.PAS) is shown in Listing 3; this and a sample program (MMTest) are included on this month's disk.

The aim is achieved by using the `Get/SetMemoryManager` routines in the `System` unit, which allows us to save and replace the default memory manager. This is done by defining a constant of type `TMemoryManager` with three procedural fields, for `Getmem`, `FreeMem` and `ReallocMem` respectively. These procedures all do essentially the same thing: first, they call the saved memory manager routines, then they save the actual heap status in the exported variable `HeapSt`, of type `THeapStatus`, write to the log file (if any) a line with the indication of the type of the action performed and the value of the allocated memory at this time and finally save this value in the private variable `OldAllocated`. Note that the write is performed only if the memory variation is larger than a threshold chosen by the user.

The `HeapSt` variable can be used to monitor the heap status while debugging. In particular, you can watch its `TotalAllocated` field to determine the total heap memory allocated at any time. Along with the `HeapSt` variable, the `MManager` unit exports three procedures: `SetDebugManager`, `ClearDebugManager` and `WriteDebug`.

`SetDebugManager` installs the new memory manager. It takes two parameters: `FName`, of type `string`, is the name of the log file (if an empty string is passed no write is performed), and `Step` represents the threshold under which a memory variation is not written to the log file. This allows you to reduce the number of lines written to the file so it doesn't get too big. After initialising some private variables, the routine sets the new memory manager and, if appropriate, rewrites the log file and writes a line with the initially allocated memory.

`ClearDebugManager` writes a line with the finally allocated memory and closes the file, then resets the default memory manager. `WriteDebug` takes a `string` as a parameter. It writes a line to the log file whenever needed, so it's easy to identify a particular situation.

You could put the `SetDebugManager` and `ClearDebugManager` routines in the `initialization` and `finalization` sections respectively of your application unit(s), or anywhere else you want, to monitor just a small section of code, as well as setting the file name and threshold anywhere you like.

Contributed by Roberto De Marini,
email: rdemari@mbox.vol.it

Combo Box Helpers

To fill combo boxes with items from a database table at run time I came up with the procedure in Listing 4, which cycles through all the components on a given form finding all the `TDBComboBox` controls. It then fills the combo boxes which have their `Tag` property set to zero with the items in the selected field from our table.

I also needed a routine which would check to see if all the `DBLookupCombo` boxes have a correct entry in them, or if they are left blank. I came up with the function in Listing 5, which cycles though the components array finding all the `TDBLookupCombo` boxes on a given form. It then fills a stringlist with the items in the `TDBLookupCombo` box `Items` list, checks the `Text` property of the `TDBLookupCombo` against the stringlist. If a valid entry was made the function returns true, or it pops up

➤ *Listing 3*

```
unit MManager;
interface
var HeapSt: THeapStatus;
procedure SetDebugManager(FName: string; AStep: integer);
procedure ClearDebugManager;
procedure WriteDebug(const S: string);
implementation
uses SysUtils;
var
  FileName : string; // the log Filename
  OldMM : TMemoryManager; // the default memory manager
  F: System.Text; // the log file
  OldAllocated : integer; //save last value of allocated memory
  Step : integer; //threshold after which we write to log file
function DebugGetMem(Size: integer): pointer;
begin
  Result:=OldMM.GetMem(Size);
  HeapSt:= GetHeapStatus;
  if (FileName <>'') and
    ((HeapSt.TotalAllocated - OldAllocated) >= Step) then
    Writeln(F,'GetMem : ',HeapSt.TotalAllocated);
  OldAllocated := HeapSt.TotalAllocated;
end;
function DebugFreeMem(P: Pointer): integer;
begin
  Result := OldMM.FreeMem(P);
  HeapSt:= GetHeapStatus;
  if (FileName <>'') and
    ((OldAllocated -HeapSt.TotalAllocated) >= Step) then
    Writeln(F,'FreeMem : ',HeapSt.TotalAllocated);
  OldAllocated := HeapSt.TotalAllocated;
end;
function DebugReallocMem(P: pointer; Size: integer): Pointer;
begin
  Result := OldMM.ReallocMem(P,Size);
  HeapSt:= GetHeapStatus;
  if (FileName <>'') and
    ((HeapSt.TotalAllocated - OldAllocated) >= Step) then
    Writeln(F,'ReallocMem : ',HeapSt.TotalAllocated);
  OldAllocated := HeapSt.TotalAllocated;
end;
const { the new memory manager }
  DebugMM: TMemoryManager = (
    GetMem : DebugGetMem;
    FreeMem : DebugFreeMem;
    ReallocMem : DebugReallocMem);
{ exported routines }
procedure SetDebugManager(FName: string; AStep: integer);
begin
  FileName := FName;
  Step:= AStep;
  OldAllocated := 0;
  if FileName <> '' then begin
    AssignFile(F,FileName);
    Rewrite(F);
  end;
  GetMemoryManager(OldMM);
  SetMemoryManager(DebugMM);
  HeapSt:= GetHeapStatus;
  if FileName <> '' then
    writeln(F,
      'Initially allocated Memory : ',HeapSt.TotalAllocated);
  OldAllocated := HeapSt.TotalAllocated;
end;
procedure ClearDebugManager;
begin
  if FileName <> '' then begin
    writeln(F,
      'Finally allocated Memory : ',HeapSt.TotalAllocated);
    CloseFile(F);
  end;
  SetMemoryManager(OldMM);
end;
procedure WriteDebug(const S: string);
begin
  if FileName <> '' then writeln(F,S);
end;
end.
```

a message and won't let the user leave that component until a valid entry is made.

Contributed by Kent Shaw, kentshaw@unitime.com

Data Validation, Required Fields & Null Values

The BeforePost event handler for TTable is the most popular place to do data validation. However, it does not catch null value entry errors for fields whose Required property is True. Typical validation code is:

```
procedure TForm1.Table1BeforePost(DataSet: TDataSet);
begin
  if DBEdit1.Text = '' then begin
    ShowMessage('Field cannot be left blank');
    DBEdit1.SetFocus;
    Abort;
  end;
end;
```

This code first checks to see if the user is trying to post a null value. If so, it tells them the field in question cannot be left blank, sets their focus on the offending edit box, and then aborts the Post procedure. However, this code does not work for a field whose Required property is set to True.

Let's use an example of DBEdit1 connected to the field ReqField, whose Required property is True. When the user tries to Post, Delphi returns the EDBEngineError message: *Field value required. Field: ReqField* and then aborts the procedure. The BeforePost event handler is completely ignored. The EDBEngineError occurs before the BeforePost event. We cannot catch the null value before it causes the error.

One solution is to never set a field's Required property to True. If you do this, your BeforePost event handler will work as expected. An exchange in the CompuServe Delphi forum indicated that this is the way most programmers deal with this problem. But it is bad programming to use this end-run around the problem. If a field requires a non-null value, the Required property should be True. Good programming practice puts data checks as deep in the application hierarchy as possible. It is better to let Delphi check on the validity of a null value than rely on your own coding.

Why? You might set the Required property to False to get around the issue, but forget to write the corresponding BeforePost event handler. Your user could wind up posting null data to an important field, possibly even a key index. On the other hand, if the Required property is set to True, the worst that can happen is that Delphi will catch the error and abort the procedure brusquely. At least this way we protect the integrity of the database.

So if the field's Required property is True, and that means the BeforePost event handler will not catch the null values, where can we catch them? The answer is to use a try...except block before you Post. Using the same example, our user tries to Post the record containing the null-value field by pressing ButtonPost. The code is shown in Listing 6.

The except clause executes if the Post fails due to any EDatabaseError, including the EDBEngineError, and we test for the null value. If the null value is what caused the error the user will see an appropriate message, have the focus set back to the offending edit box, and the procedure aborts. You need to place similar code in every control which triggers a Post, including controls that call an implied Post (eg move to next record or insert a record).

```
procedure ReadTable(Sender: TObject);
var
  j: integer;
  code : String;
  test : Boolean;
begin
  for j := 0 to ComponentCount - 1 do begin
    if Components[j] is TDBComboBox then begin
      if (Components[j] as TDBComboBox).tag = 0 then begin
        Table.Open;
        Table.First;
        while not Table.EOf do begin
          (Components[j] as TDBComboBox).Items.Add(
            Table.FieldByName('CODE').AsString);
          Table.next;
        end;
        Table.Close;
      end;
    end;
 end;
end;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
function ChkEntries(Sender: TObject):Boolean;
var j,i,k: integer;
    page: string;
begin
  for i := 0 to ComponentCount - 1 do
    if ((Components[i] is TControl) and
      ((Components[i] as TControl).parent =
      notebook.pages.objects[PageIndex])) then
      for j := 0 to ComponentCount - 1 do begin
        if ((Components[j] is TControl) and
          ((Components[j] as TControl).parent =
          components[i])) then begin
          if Components[j] is TDBLookupCombo then begin
            stringlist.Clear;
            stringlist.AddStrings(
              (Components[j] as TDBLookupCombo).items);
            if not stringlist.Find(
              (Components[j] as TDBLookupCombo).text, k)
              then begin
              if ((Components[j] as TDBLookupCombo).text)
                <> '' then begin
                MessageDlg('Invalid, Please Re-Enter',
                  mtInformation, [mbOK], 0);
                (Components[j] as TDBLookupCombo).text :=
                  '';
                ActiveControl :=
                 (Components[j] as TDBLookupCombo);
                Result := false;
              end;
            end else
              Result := true;
          end;
        end;
      end;
end;
```

Contributed by Glen Janken, gjanken@pqsoft.com

➤ *Listing 6*

```
procedure TForm1.ButtonPostClick(Sender: TObject);
begin
  try
    Table1.Post;      {try to Post}
  except
    on EDatabaseError do begin
      { We didn't get to the BeforePost event handler
        before the error occurred }
      if DBEdit1.Text = '' then begin
        ShowMessage('This Field cannot be left blank');
        DBEdit1.SetFocus;
        Abort;
      end;
    end;
  end;
end;
```