The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

Radio Group Problem

Q I regularly use `TRadioGroup` components, but have a gripe with them, which also applies to `TDBRadioGroup` controls. When I change the `Cursor` property, I am only able to change the cursor of the group itself and not the constituent radio buttons. I wouldn't mind, but I don't seem to be able to access the radio buttons individually to change their cursors. Is there any under-the-hood way of approaching the problem?

A Yes, in a word. You can loop through the component's `Controls` property, which is a list of the children of the given component. It is a good idea to check the type of each component found to ensure that it really is a `TRadioButton`, but given that, you can then change the `Cursor` property. This can be done explicitly in your code, or you could make new components to automate the process.

The project RBCur.Dpr on the disk employs both approaches (so you will need to install the components before loading the project). The project has a Delphi-supplied radio group and data-aware radio group and also a new version of each component (`TNewRadioGroup` and `TDBNewRadioGroup` as defined in RGroup.Pas). A button causes the cursor to be changed on all these components, ensuring that it is changed not just for the group but also for the constituent radio buttons. Listing 1 shows a snippet that works with a radio group.

The business of 'componentising' this functionality isn't as simple as it might first appear. When the `Cursor` property gets assigned to, the private, non-virtual `TControl.SetCursor` method gets called. Since it is non-virtual it cannot be overridden, but fortunately the method sends the underlying control a `cm_CursorChanged` message. We are more than able to trap this message and do whatever is necessary. Upon first glance, it would seem that in the message handler we should loop through the controls and set the cursor much as in Listing 1. However, this is not quite good enough, for a reason that I first observed once upon a time with `TDataSet` derivatives.

```
procedure SetRadioGroupCursor (RG: TCustomRadioGroup; Cur: TCursor);
var Loop: Integer;
begin
  with RG do begin
    Cursor := Cur;
    for Loop := 0 to ControlCount - 1 do
      if Controls[Loop] is TRadioButton then
        TRadioButton(Controls[Loop]).Cursor := Cur
  end
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  { Set Cursor property of radio buttons in radio group }
  SetRadioGroupCursor (RadioGroup1, crHandPoint);
  { Set Cursor property of radio buttons in db radio group }
  SetRadioGroupCursor (DBRadioGroup1, crSQLWait);
  ...
end;
```

➤ *Listing 1*

Consider a form that has been set up at design-time with a `TTable` on it. The table has its `DatabaseName` and `TableName` property set up to some valid values and the `Active` property is set to `True`. Setting the `Active` property to `True` calls the internal `SetActive` method which tries to open specified table in the specified database. Having got that idea firmly in mind, now consider what happens when this form is created at runtime. As the form resource is processed, each component is created in turn and each property value is individually read in from the resource and set. There is no guarantee in which order those properties are stored in the form file. If the `Active` property is retrieved before the `DatabaseName` and `TableName` properties, then it would be a pointless exercise trying to open anything.

This is where the `Loaded` method comes into play. When a component has had all its properties read in, the `Loaded` method is called. This allows any final actions that couldn't be performed *during* the property reading to be taken care of. If a value of `True` for the `Active` property is encountered during the process of reading the `TTable` from a form file, it sets an internal `Boolean` flag. The `Loaded` method checks the flag, and if necessary performs the real table opening operation.

The same principle applies with this radio group cursor problem. If the `Cursor` property is read from the form file before all the radio button captions have been read (and therefore before the radio button objects have been manufactured) it is pointless trying to set all their cursors. Instead, the component's `Loaded` method does that by again sending a `cm_CursorChanged` message to itself.

I was going to reuse the `SetRadioGroupCursor` routine from Listing 1 to do the radio button

cursor setting, but there would have been a problem with that. You will see that the routine sets the radio group `Cursor` property, which calls `TControl.SetCursor`, which sends a `cm_CursorChanged` message, which my component traps, which would then call `SetRadioGroupCursor` and in turn that would set the `Cursor` property, etc. Unsightly recursion would ensue and that simply would not do. So, a slightly modified version of the code is used, since it is called in response to the component's cursor changing, the explicit cursor change has been removed.

Listing 2 shows one of the two components (the other is practically the same). The source code shows that you can tell if your component properties are still being read from a form stream file by seeing if the `csReading` value is in the `ComponentState` set property.

```
type
  TNewRadioGroup = class(TRadioGroup)
  private
    FStreamedCursor: Boolean;
  protected
    procedure CMCursorChanged(var Msg: TMessage);
      message cm_CursorChanged;
    procedure Loaded; override;
  end;
...
procedure SetRadioGroupCursor(RG: TCustomRadioGroup; Cur: TCursor);
var
  Loop: Integer;
begin
  with RG do
    for Loop := 0 to ControlCount - 1 do
      if Controls[Loop] is TRadioButton then
        TRadioButton(Controls[Loop]).Cursor := Cur
end;
procedure TNewRadioGroup.CMCursorChanged(var Msg: TMessage);
var
  Loop: Integer;
begin
  inherited;
  if csReading in ComponentState then
    FStreamedCursor := True
  else
    SetRadioGroupCursor(Self, Cursor)
end;
procedure TNewRadioGroup.Loaded;
begin
  inherited Loaded;
  if FStreamedCursor then begin
    FStreamedCursor := False;
    Perform(cm_CursorChanged, 0, 0)
  end
end;
```

➤ *Listing 2*

Application Time-Out

Q I have a requirement to shut down a Delphi application after a fixed time during which there is no keyboard or mouse action. Any action should reset the delay. It should work whichever form has focus or, indeed, even if the application has lost focus.

A This sounds like it might be to make 'nobbled' demo versions of software. The basic requirement is much like the mentality of a screen saver. After a certain period of inaction the system will launch the appropriately specified screen saver, and it does this by using system-wide hooks plugged into the mouse and keyboard events. For your application, you will need only thread-wide hooks for these events. In a Delphi application, all the standard user interface work is done in the primary thread of the application.

In Issue 26, Warren Kovach demonstrated how to implement system-wide hooks so we don't need to go over too much ground here *[but, be sure to check out Warren's update in Issue 32. Ed]*. Basically we will need to call `SetWindowsHookEx` once for the mouse events and once for the keyboard events. For each call, we need a hook routine that will re-set a timer each time any mouse or keyboard events fire. At the end of the program we will need to uninstall these events. The code will look like Listing 3 (taken from the AppDie.Dpr project).

```
var
  hkKb, hkMouse: HHook;
function HookProc(HHook, Code: Integer; WParam: Cardinal; LParam: Longint):
  Longint;
begin
  Form1.Timer1.Enabled := False;
  Form1.Timer1.Enabled := True;
  Result := CallNextHookEx(HHook, Code, WParam, LParam)
end;
function HookProcKB(Code: Integer; WParam: Cardinal; LParam: Longint):
  Longint; stdcall;
begin
  Result := HookProc(hkKB, Code, WParam, LParam)
end;
Function HookProcMouse(Code: Integer; WParam: Cardinal; LParam: Longint):
  Longint; stdcall;
begin
  Result := HookProc(hkMouse, Code, WParam, LParam)
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  hkKb := SetWindowsHookEx(wh_Keyboard, @HookProcKB, 0, GetCurrentThreadID);
  hkMouse := SetWindowsHookEx(wh_Mouse, @HookProcMouse, 0, GetCurrentThreadID);
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  UnhookWindowsHookEx(hkKb);
  UnhookWindowsHookEx(hkMouse);
end;
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Close;
end;
```

➤ *Listing 3*

There are a number of things to observe about this code. The hook procedures that are logged with the system must be declared `stdcall`. For the program to act normally, you must chain on to the original hook routine with `CallNextHookEx`. Also, to install a hook just for the current thread,

```
const
  DefaultTimeOut = 10000;
type
  TTimeOut = class(TComponent)
  private
    FTimeOut: Cardinal;
  protected
    procedure SetTimeOut(Value: Cardinal);
    procedure TimerTick(Sender: TObject);
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property TimeOut: Cardinal
      read FTimeOut write SetTimeOut default DefaultTimeOut;
  end;
const Count: Byte = 0;
constructor TTimeOut.Create(AOwner: TComponent);
begin
  if Count > 0 then
    raise Exception.Create(
      'Only one of these time-out components allowed');
  Inc(Count);
  inherited Create(AOwner);
  FTimeOut := DefaultTimeout;
  if not (csDesigning in ComponentState) then begin
    Timer := TTimer.Create(Self);
    Timer.OnTimer := TimerTick;
    Timer.Interval := FTimeOut;
    hkKb := SetWindowsHookEx(wh_Keyboard, @HookProcKB, 0,
      GetCurrentThreadID);
    hkMouse := SetWindowsHookEx(wh_Mouse, @HookProcMouse, 0,
      GetCurrentThreadID);
  end;
end;
procedure TTimeOut.SetTimeOut(Value: Cardinal);
begin
  if FTimeOut <> Value then begin
    FTimeOut := Value;
    Timer.Interval := Value;
  end
end;
procedure TTimeOut.TimerTick(Sender: TObject);
begin
  Application.Terminate
end;
```

➤ *Listing 4*

you pass its thread identifier (obtained with `GetCurrentThreadId`) to `SetWindowsHookEx`.

The code on the disk has conditional compilation to make it work for all Delphi versions. The main differences for Delphi 1 are the use of the `far` keyword instead of `stdcall` and `GetCurrentTask` instead of `GetCurrentThreadId`. Also, in 16-bit, you do not need to take the address of the hook routine, so use of the `@` operator disappears. One final 16-bit issue is that a `TTimer` component's `Interval` property has a maximum value of 65535, limiting the time-out to 65½ seconds.

Of course one obvious improvement that could be made to this code would be to turn it into a component. The file TimeOut.Pas has a possible implementation, which is used in the second project AppDie2.Dpr. Again, this has been written to be platform-independent. A lot of the code is the same, but it takes care only to execute the time-out code if running outside the development environment. Also it must manufacture the `TTimer` object on its own. Additionally, since some global variables are used to store the hook handles, the code tries to ensure only one instance of the component can be used in any application. Some of the important bits of code can be seen in Listing 4.

Event Complexity

Q I often have to change the state of checkboxes, radio groups, listboxes and so on. The problem is that when I do something like that under program control, this very action also triggers the `OnClick` or `OnChange` events of the control I'm changing. This is generally not what I want. Right now I'm using `Boolean` flags to dictate when the event handler code should execute and when not. As you can probably imagine, this is rather cumbersome in large programs because all kinds of events trigger other events. Is there some means to overcome this problem or to solve it in a nicer way?

A I can think of two approaches to this problem. Your suggestion is one solution, where the event handlers make sure they only execute their own code when appropriate, as dictated by flags. This involves writing lots of conditional code inside your event handlers. A simple project called Event.Dpr

➤ *Listing 5*

```
var
  IgnoreCheck: Boolean;
procedure TForm1.Button1Click(Sender: TObject);
begin
  IgnoreCheck := True;
  try
    CheckBox1.Checked := not CheckBox1.Checked
  finally
    IgnoreCheck := False
  end
end;
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if not IgnoreCheck then
  begin
    ShowMessage('The checkbox was clicked by the user');
    { Blah, blah, blah }
  end
end;
```

➤ *Listing 6*

```
procedure TForm1.Button1Click(Sender: TObject);
var
  OldHandler: TNotifyEvent;
begin
  OldHandler := CheckBox1.OnClick;
  CheckBox1.OnClick := nil;
  try
    { This won't trigger the event since it has been disabled }
    CheckBox1.Checked := not CheckBox1.Checked
  finally
    CheckBox1.OnClick := OldHandler
  end
end;
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  ShowMessage('The checkbox was clicked by the user');
  { Blah, blah, blah }
end;
```

shows this idea, and some code from it can be seen in Listing 5.

An alternative that springs to mind would involve disabling the event handler for as long as is necessary and then re-enabling it afterwards. You can disable an event handler by assigning `nil` to it, and you can save the old handler in an appropriately typed variable. The online help, or VCL source code, can help identify what the appropriate event type would be. Some code that does this is in Listing 6 (from Event2.Dpr).

Following a desire to make this endeavour slightly less repetitive, to cut down on the variable declarations, I produced Listing 7 (from Event3.Dpr). The idea is to have a helper routine that stores the event details. To do this I am using a typed constant (otherwise known as an initialised static variable) of the generic `TMethod` type with event type typecasts where necessary. The problem here is that the syntax is not liked by Delphi 1. So, if you are using Delphi 2 or 3, you could take this approach.

Since the calls to `SetHandler` in Listing 7 ends up being a little overbearing, we now have the final offering. Listing 8 comes from Event4.Dpr. This code works in all versions of Delphi (so far), and uses runtime type information (RTTI) to disable and reset the event handler. The details of how the RTTI stuff operates are not strictly that important, but basically the same things happen as with Listing 7. Having written the routine once, the business of disabling and re-enabling the event handler becomes a little simpler.

```
const
  NilMethod: TMethod = (Code: nil; Data: nil);
function SetHandler(Handler: TMethod): TMethod;
const
  OldHandler: TMethod = (Code: nil; Data: nil);
begin
  Result := OldHandler;
  OldHandler := Handler
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  CheckBox1.OnClick := TNotifyEvent(SetHandler(TMethod(CheckBox1.OnClick)));
  try
    { This won't trigger the event since it has been disabled }
    CheckBox1.Checked := not CheckBox1.Checked
  finally
    CheckBox1.OnClick := TNotifyEvent(SetHandler(NilMethod))
  end
end;
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  ShowMessage('The checkbox was clicked by the user');
  { Blah, blah, blah }
end;
```

➤ *Listing 7*

```
uses
  TypInfo;
procedure SetHandler(AObject: TObject; Event: String; Enable: Boolean);
const
  { Used for saving old event handler }
  OldHandler: TMethod = (Code: nil; Data: nil);
  { Used for disabling event handler }
  NilMethod: TMethod = (Code: nil; Data: nil);
var
  PropInfo: PPropInfo;
begin
  PropInfo := GetPropInfo(AObject.ClassInfo, Event);
  if not Assigned(PropInfo) then
    raise Exception.CreateFmt('Event %s not found in %s class',
      [Event, AObject.ClassName]);
  with PropInfo^ do begin
    if PropType^.Kind <> tkMethod then
      raise Exception.CreateFmt('%s is not an event', [Event]);
    if Enable then
      SetMethodProp(AObject, PropInfo, OldHandler)
    else begin
      OldHandler := GetMethodProp(AObject, PropInfo);
      SetMethodProp(AObject, PropInfo, NilMethod)
    end
  end
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  SetHandler(CheckBox1, 'OnClick', False);
  try
    { This won't trigger the event since it has been disabled }
    CheckBox1.Checked := not CheckBox1.Checked
  finally
    SetHandler(CheckBox1, 'OnClick', True);
  end
end;
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  ShowMessage('The checkbox was clicked by the user');
  { Blah, blah, blah }
end;
```

➤ *Listing 8*

Updating System Files

Q As described in *The Delphi Magazine* Issue 31's *Clinic*, I encountered the Delphi 3 display glitch too. I have now got a copy of ComCtl32.DLL v4.71 but am unable to install this version over the current one (v4.70). I am running Windows NT 4 and when I try to copy it, I get an error dialog telling me the file is in use. Booting from a disk to solve this doesn't work for me, because my system partition is NTFS. Do you have any thoughts on how to solve this?

A This question opens up a standard problem of how to install a file that Windows currently has in use.

Windows 95 users can usually do this, as they can boot into DOS and then do a manual file copy, but to cater for NT users, and also to cater for automating the process, we need more. Unfortunately, the approach required is inconsistent: it differs for Windows 95 and Windows NT.

With Windows NT, you can write a small program to do the job, using the `MoveFileEx` Windows API call. Assuming that you have the newer library file stored as

```
C:\Windows\System32\ComCtl32.New
```

then a call like this should do it:

```
MoveFileEx(
  'C:\Windows\System32\ComCtl32.New',
  'C:\Windows\System32\ComCtl32.Dll',
  MoveFile_Delay_Until_Reboot)
```

The help for `MoveFileEx` says it operates by storing relevant information in the registry which is processed whilst NT is initialising.

For Windows 95, the `MoveFile_Delay_Until_Reboot` flag is not supported, so we need to make use of WinInit.Exe. This is the program that produces the occasional Windows 95 start-up messages that you may have seen, like:

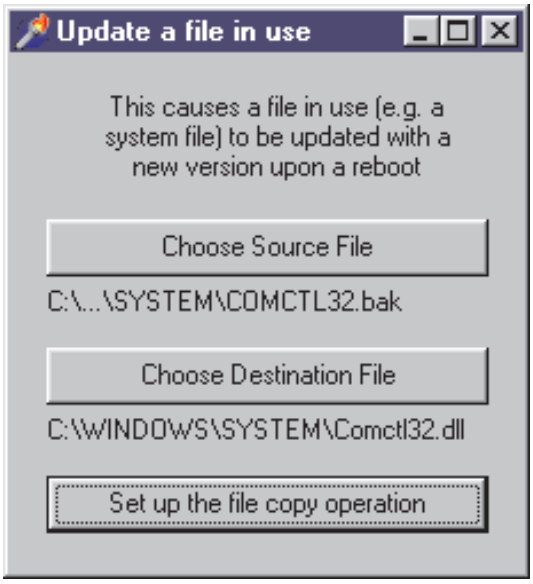
> *Please wait while Setup updates your configuration files.*
>
> *This may take a few minutes...*
>
> *Completed updating files, continuing to load Windows...*

`WinInit` operates using an INI file called WinInit.Ini in the Windows directory. If it exists, `WinInit` will process the file during start-up, before Windows switches to protected mode, and then it will rename it with a .BAK extension. An appropriate looking section from this INI file that will do the job would be:

```
[rename]
C:\windows\system\comctl32.dll=
  c:\windows\system\comctl32.new
```

The sample project on the disk UpdateSystemFile.Dpr uses a few components to simplify setting these restart options (see Figure 1). Listing 9 contains the code for the three buttons. Notice that care is taken to ensure short file and path names are used for the `WinInit` approach. Since WinInit executes before the long filename management code is running, short filenames are mandatory. Additionally, the handy `MinimizeName` routine from the `FileCtrl` unit is employed to make sure the filename fits nicely onto the form. If the filename would be too long, an ellipsis (...) is employed to replace the excess characters. You can see this in the source file name in Figure 1.

➤ *Figure 1*



But now, with all that said, I can offer a potentially simpler solution in the case of the common control library. The following web page has a utility program that will do the job for you:

```
www.microsoft.com/msdn/
  downloads/files/
  40comupd.htm
```

40ComUpd.Exe (which is about 500Kb) can be downloaded from here and, when executed, will update your library to version 4.72. It almost certainly does what is described above and means that if you run it and then restart Windows, you will have the newer library without any fuss and bother. However, the technique has many more applications, of course, not least in installer utilities.

For more information on these file update techniques, refer to article Q140570 on the Microsoft Developer Network Library CD or the appropriate area on the Microsoft website. Article Q172456 also describes how to use .INF files to set the files up in the first place. This is explored in more detail in Appendix C of the Microsoft Windows 95 Resource Kit.

Acknowledgements

Thanks are due to Sandy McCourt for the original source code that solves the application time-out problem. Thanks to Jack Bakker for locating the appropriate Microsoft web page for the common control library update.

➤ *Listing 9*

```
uses FileCtrl, IniFiles;
procedure TForm1.btnChooseSrcClick(Sender: TObject);
begin
  if dlgSrc.Execute then begin
    lblSrc.Caption := MinimizeName(dlgSrc.FileName, Canvas, btnChooseSrc.Width);
    btnChooseDst.Enabled := True;
    btnChooseDst.SetFocus
  end
end;
procedure TForm1.btnChooseDstClick(Sender: TObject);
begin
  dlgDst.InitialDir := ExtractFilePath(dlgSrc.FileName);
  if dlgDst.Execute then begin
    lblDst.Caption := MinimizeName(dlgDst.FileName, Canvas, btnChooseDst.Width);
    btnSetup.Enabled := True;
    btnSetup.SetFocus
  end
end;
procedure TForm1.btnSetupClick(Sender: TObject);
var
  Src, Dst: array[0..MAX_PATH] of Char;
begin
  case Win32Platform of
    VER_PLATFORM_WIN32_WINDOWS :
      with TIniFile.Create('WinInit.Ini') do
        try
          GetShortPathName(PChar(dlgSrc.FileName), Src, MAX_PATH);
          GetShortPathName(PChar(dlgDst.FileName), Dst, MAX_PATH);
          WriteString('rename', Dst, Src)
        finally
          Free
        end;
    VER_PLATFORM_WIN32_NT :
    begin
      Win32Check(MoveFileEx(PChar(dlgSrc.FileName),
        PChar(dlgDst.FileName),
        MoveFile_Delay_Until_Reboot))
    end
  end
end;
```