The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

Just email Brian Long, our Delphi Clinic Editor, on blong@compuserve.com or write/fax us at The Delphi Magazine

SDI Menu Problems

Q If I make an SDI application with menus on several forms, I seem to walk into a problem. If I try and choose a menu item (using the keyboard) the keystroke is sent to the main form. For example, if I try and bring down my `Font` menu on a form other than the main form, the main form's `File` menu drops down. I can see that Delphi itself takes advantage of this so that from the form designer you can pull down the main form's menus, but I want extra menus and I want them to work.

A In investigating this problem I have come across a number of aspects to it. Firstly, your current applications' menus can be accessed via the keyboard if you do things one keystroke at a time. In other words, instead of pressing `Alt-F`, press `Alt` then press `F` separately. Also, the problem doesn't occur if the secondary form has no controls on it. Lastly, this was clearly a bug since Delphi 3 fixes it.

The problem occurs because code in the VCL tries to make SDI applications operate rather like MDI applications in general use. In other words, close the main form and the whole application closes, press a menu keystroke on a non-main-form and the main form's menu comes to life and so on. However, the coding in Delphi 1 and 2 was a little over-zealous and keeps this menu transference up even if there is a menu on the non-main-form.

When any form gets a `wm_SysCommand` indicating that a keystroke was used to pull down a menu, it immediately sends a `cm_AppSysCommand` message to the `Application` object. `Application`'s window procedure then sends the menu-invoking keystroke to the main form which causes the problem. Delphi 3 fixes this by only doing this redirection if the original form has no menu.

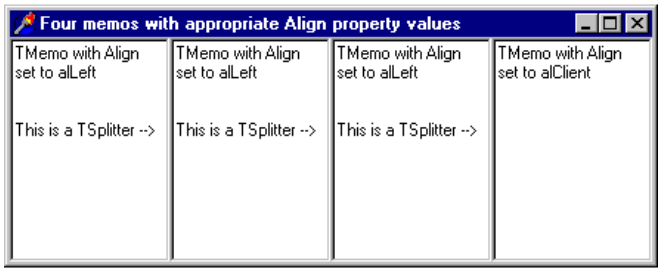
To stop the redirection, you need to trap the `cm_AppSysCommand`

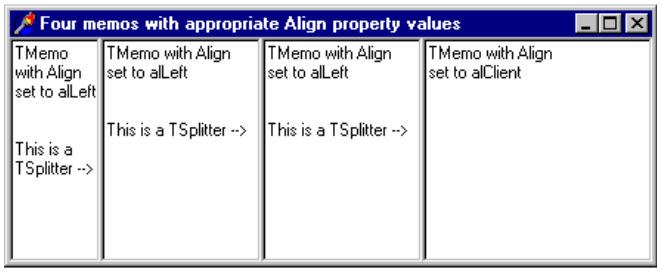
➤ *Listing 1*

```
TMainForm = class(TForm)
...
  procedure FormCreate(Sender: TObject);
  procedure FormDestroy(Sender: TObject);
private
  function DoAppMessage(var Msg: TMessage): Boolean;
end;
...
function TMainForm.DoAppMessage(var Msg: TMessage): Boolean;
begin
  Result := (Msg.Msg = cm_AppSysCommand)
{$ifndef CompletelyStopMenuRedirections}
    and (Screen.ActiveForm.Menu <> nil)
{$endif}
end;
procedure TMainForm.FormCreate(Sender: TObject);
begin
  Application.HookMainWindow(DoAppMessage);
end;
procedure TMainForm.FormDestroy(Sender: TObject);
begin
  Application.UnHookMainWindow(DoAppMessage);
end;
```

➤ *Listing 2*

```
unit MenuFix;
interface
implementation
uses SysUtils, Forms, Messages, Controls;
type
  TFixMyMenus = class
  private
    function DoAppMessage(var Msg: TMessage): Boolean;
  public
    constructor Create;
    destructor Destroy; override;
  end;
constructor TFixMyMenus.Create;
begin
  inherited Create;
  Application.HookMainWindow(DoAppMessage)
end;
destructor TFixMyMenus.Destroy;
begin
  Application.UnhookMainWindow(DoAppMessage);
  inherited Destroy
end;
function TFixMyMenus.DoAppMessage(var Msg: TMessage):
  Boolean;
begin
  Result := (Msg.Msg = cm_AppSysCommand)
{$ifndef CompletelyStopMenuRedirections}
    and (Screen.ActiveForm.Menu <> nil)
{$endif}
end;
const Obj: TFixMyMenus = nil;
procedure MyExitProc; far;
begin
  Obj.Free
end;
{$ifdef Ver80} {Delphi 1.0x}
  {$define DodgyMenus}
{$endif}
{$ifdef Ver90} {Delphi 2.0x}
  {$define DodgyMenus}
{$endif}
{$ifdef Ver93} {C++ Builder 1.0x}
  {$define DodgyMenus}
{$endif}
initialization
{$ifdef DodgyMenus}
  AddExitProc(MyExitProc);
  Obj := TFixMyMenus.Create
{$endif}
end.
```





➤ *Above Left: Figure 1*

➤ *Above Right: Figure 2*

```
procedure TForm1.Memo1DragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  { Allow edit -> memo drag & drop }
  Accept := Source is TLabel
end;
procedure TForm1.Memo1DragDrop(Sender, Source: TObject; X, Y: Integer);
var
  Pt: TSmallPoint;
begin
  with Sender as TMemo do begin
    { Turn co-ordinates into a TSmallPoint }
    Pt := PointToSmallPoint(Point(X, Y));
    { Do a down click }
    Perform(wm_LButtonDown, 0, Longint(Pt));
    { Do an up click }
    Perform(wm_LButtonUp, 0, Longint(Pt));
    { The memo's input caret is now nicely poised }
    { Insert the edit's selected text into the memo }
    SelText := (Source as TLabel).Caption
  end
end;
```

➤ *Listing 3*

and stop it getting to the `Application`'s window procedure. If the message was posted, we could use `Application`'s `OnMessage` event. However, since it is sent, we must use `HookMainWindow` to trap it. This requires a function method that takes a `TMessage` record as a `var` parameter and returns a `Boolean`. If the return is `True` the message does not get to `Application`, if the return is `False` it does.

Listing 1 shows some code from the main form unit (of the project MenuFixP.Dpr in the Clinic\Menus1 directory on the disk) that sets such a routine up. Note that there is some conditional compilation going on. If the conditional symbol `CompletelyStopMenuRedirections` is not defined (the default situation) then the main form will still pick up menu keystrokes if the active form has no menu set up. If you define the symbol, then the application will function in a more natural Windows-like way. That is, the main form will never pick up any other form's keystrokes.

To help cater for already finished applications and also to help people with Borland C++ Builder, I have supplied a second solution, which is a self-contained unit that can simply be added into a Delphi or C++ project to remedy the problem (Listing 2). This has the benefit that (providing the source code is available) conditional symbols can ensure the code is only compiled for the appropriate product versions (ie not Delphi 3 or higher). The project in the Menus2 subdirectory from Clinic on this month's disk employs this solution, relieving the main form of doing anything special.

Drag And Drop Onto TMemo

Q In my application I need to support drag and drop where the target can be a `TMemo`. When the user makes the drop, I want to insert some text in the memo at the point where the mouse cursor is. How can I work out where to insert the text when all I have is an X and Y position?

A It seems to me that there are two ways of dealing with this problem. One is rather tricky and involves checking the font in the memo, using the font's metrics to identify where to insert text, taking into account multiple lines and other complexities. The easier way is to let the memo do all the hard work. Bear in mind that when you click on a memo, the input caret is placed at the sensible and appropriate position. We can do a fake mouse click when the user does the drop and the same thing will happen. The MemoText.Dpr project on this month's disk shows this. There is a label and a memo. You can drag from the label into the memo. The label's caption gets inserted where the mouse was when the drop occurred. Listing 3 shows the relevant event handlers.

Splitter Question

Q I am trying to use the new Delphi 3 splitter component in a certain way but am failing. I have some controls set up with splitters in between, rather like the memos I have in the screen shot (Figure 1). If I drag the first splitter left, the second and third splitters also move left by the same amount. This means that the second and third memos stay the same width, but the fourth gets wider as shown in my other screen-shot (Figure 2). What I want is for the second and third splitter to remain exactly where they are, resulting in the second memo getting larger, and the third and fourth remaining the same. Can this be done?

A It can. Your project has been saved on the disk as SplitterProblem.Dpr. Another one, SplitterSolution.Dpr, does what you want (see Figure 3). To set it up is quite tricky to explain, but I'll do my best. If it ends up being difficult to understand, load up the project and view the main form as text (`Alt-F12`).

Set down a master panel object (`Panel1`) with whatever size or alignment is required. Now put another panel (`Panel2`) inside the first one, about three quarters the size, and set its `Align` property to `alLeft`. Next, place a splitter next to `Panel2` inside `Panel1` also aligned to the left and then a memo next to the splitter in `Panel1`, aligned to client (this will be the rightmost memo). Now we go through the same process in `Panel2`. Place another panel (`Panel3`) in `Panel2`, align it to the left. Then place a splitter next to it, aligned to the left and a memo next to that, aligned to client. That leaves two memos to go. Place one memo in `Panel3`, aligned to the left, then a splitter next to it aligned to the left and finally the last memo inside `Panel3` aligned to client.

Once you get the idea, you can go back and customise the panels and splitters as required.

Generalised Minimising

Q When you minimise the main form of a Delphi application (including Delphi itself) all visible forms (including already minimised ones) are shrunk down into one application icon. When you minimise a non-main form, it just minimises as normal. I have reasons for wanting all forms in my Delphi project to act like the main form when they are minimised. This comes in handy for modal forms which have minimise buttons on, given that other forms in the project are disabled whilst a modal one is being displayed. How do I do this?

A The way the main form does its business is that when it is minimised, it calls `Application.Minimize`. That hides all visible forms and displays only one icon. The fact that all forms are hidden rather than minimised explains why you don't see the standard Windows 95 or Windows NT animation that normally accompanies minimising windows. The one icon is the `Application` object's window.

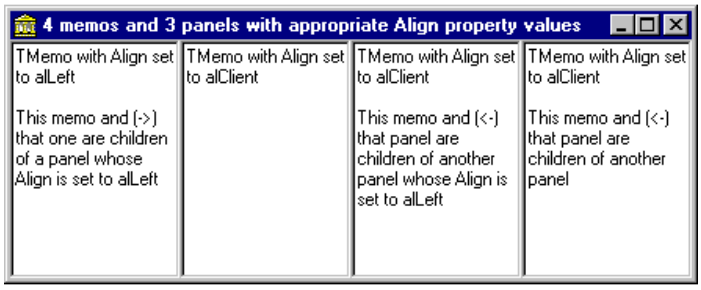
To achieve your goal, you will need to make all other forms act like the main form in this regard. In Delphi 2 or 3 you can put the required functionality into one form and use form inheritance to propagate it to all the others. In Delphi 1 you will need to replicate it through each non-main form.

Every form in question needs a `wm_SysCommand` message handler to trap the minimisation and call `Application.Minimize`. An event handler also needs to be set up for the `Application`'s `OnRestore` handler to ensure the appropriate form is left with focus when the program is restored from iconic state.

A sample program that does this is supplied as GenMin.Dpr. Listing 4 shows the main form with the `OnRestore` handler and Listing 5 has the non-main-form with its message handler. Notice that the form that was focused is stored in a global variable in the `GenVars` unit, defined as:

```
var MinForm: TForm = nil;
```

➤ *Figure 3*




```
TForm1 = class(TForm)
  procedure FormCreate(
    Sender: TObject);
private
  procedure DoRestore(
    Sender: TObject);
end;
...
procedure TForm1.DoRestore(
  Sender: TObject);
begin
  if Assigned(MinForm) then begin
    MinForm.BringToFront;
    MinForm := nil
  end
end;
procedure TForm1.FormCreate(
  Sender: TObject);
begin
  Application.OnRestore := DoRestore
end;
```

➤ *Listing 4*

```
TForm2 = class(TForm)
private
  procedure WMSysCommand(
    var Msg: TWMSysCommand);
    message wm_SysCommand;
end;
...
procedure TForm2.WMSysCommand(
  var Msg: TWMSysCommand);
begin
  if Msg.CmdType and
    $FFF0 = sc_Icon then begin
    Application.Minimize;
    MinForm := Self
  end else
    inherited
end;
```

➤ *Listing 5*

Query Execution Speed

Q I have an SQL expression that runs across Paradox tables. In Delphi 1 it took only a couple of seconds to execute. In Delphi 2 it takes about 40 seconds. This slowdown is unacceptable, what can I do about it?

A The local SQL engine in the 16-bit BDE was implemented on top of the QBE engine. SQL was translated to QBE and then executed. In the 32-bit BDE, SQL is parsed and executed natively using the new SQL interpreter. In many cases, the performance is much the same, but there are situations where things take longer. Clearly you have walked into one. From my experience, I would recommend changing your SQL query into QBE and executing that instead. In Delphi 1, executing a .QBE file was no more difficult than using a `TTable` object and setting the `TableName` property to point to it. Delphi 2 stops this undocumented support, which makes things harder. You can use third party components to manipulate QBE files: InfoPower has a QBE component that lets you use parameters etc. The time I saw the problem, executing the QBE file gave performance similar to that found in Delphi 1.

Acknowledgements

Thanks to Steve Axtell from Borland's European Technical Team for help with some of this issue's entries.