The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

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

Task Bar Calculations

Q I have a problem with the Windows task bar. My application has a status bar which needs to sit at the bottom of the screen. If the task bar is there it hides my status bar. How do I find out whether it is there, and its position?

A According to the Windows API help file, it seems that `SystemParametersInfo` is the routine for the job. Amongst many other jobs, it can tell you the available working area on your primary Windows monitor. The working area is the portion of the screen not obscured by the task bar and system tray. Note that if the task bar has not been set to stay on top, or is set to auto-hide, it is not considered to obscure any part of the screen. To demonstrate the point, you could put the code in Listing 1 into a form's `OnCreate` event handler and it would make the form take up all the screen space left by the task bar.

IDE Dissatisfaction

Q The Code Completion window in Delphi 3 and 4 seems to be a specific fixed width. If it displays something that cannot be entirely seen, holding the mouse over it produces a tooltip that contains the partially obscured text in its entirety. However the tooltip only lasts for about 2 seconds (which doesn't seem long enough). Tooltips in Microsoft applications have a tendency to be shown for longer. Can this be changed in Delphi?

A Microsoft applications seem to display their tooltips for 8 seconds. Delphi 1's tooltips stayed visible until you moved the mouse away, but the Delphi 2 (and above) default is 2.5 seconds. This is dictated by the private `DefHintHidePause` constant in the `implementation` part of the `Forms` unit. However, a Delphi application can change this default by assigning a different number of milliseconds to the `Application`'s `HintHidePause` property.

You can do the same thing for Delphi itself (which, remember, is a Delphi application) by installing a unit (as if it contained a component) that contains an assignment to `Application.HintHidePause`. To ensure Delphi compiles that code in and executes it, the statement should be contained either within the unit's `initialisation` section, or inside the `Register` routine. So either of the two units in Listings 2 and 3 would do the trick.

To install the unit once you have made it, choose `Component | Install component...`, then browse and find the unit. Also choose some package to install it into (the default Delphi Users Components package would suffice). Then press `OK` and the package will recompile. Delphi will load the package and the statement will be executed (see Figure 1). Case closed.

More On Property Editors

Q I am trying to create a new component like a `TDBEdit`, but with two data sources and two `DataField` properties.

When I create a property `DataField` I can choose field names in the Object Inspector, just as with a Delphi-supplied data-aware control. However, when I create a property `DataField2` it doesn't offer a value list. Somehow there is a link with the name `DataField` and `DataSource` that gives a value list in the Object Inspector. How does it work?

```
var
  Rect: TRect;
...
  SystemParametersInfo(
    spi_GetWorkArea, 0, @Rect, 0);
  with Rect do
    SetBounds(Left, Top,
      Right-Left, Bottom-Top);
```

➤ *Listing 1*

```
unit HintSetU;
interface
implementation
uses
  Forms;
initialization
  Application.HintHidePause := 8000
end.
```

➤ *Listing 2*

```
unit HntSetU2;
interface
procedure Register;
implementation
uses Forms;
procedure Register;
begin
  Application.HintHidePause := 8000
end;
end.
```

➤ *Listing 3*

A The VCL has registered a custom property editor to cater for all data-aware controls. All these components have a property `DataField`. The custom property editor has been registered for any `String` property called `DataField` in any component inherited from `TComponent`. Your second property is not called `DataField` and so does not get this functionality. This means you will have to provide it yourself.

The code used by the IDE can be found in a source file called DBREG.PAS in the LIB directory off the root installation directory of

Delphi 1 and 2, but it is not supplied with Delphi 3 and 4 (which has this unit contained within a design-time package).

Listing 4 shows some code from a unit DataFld2.Pas that implements and registers an appropriate list-generating property editor, with code ripped and modified from the aforementioned Delphi-supplied file. You can take this unit and install it in exactly the same way as you install a component in any version of Delphi. This will then make sure that any `String` property called `DataField2` in any registered component class directly or indirectly inherited from `TComponent` uses this property editor and gives you the desired effect.

To understand how the property editor works and get more information on the subject, you should refer back to previous articles. Some examples include:

Under Construction: Property Editors by Bob Swart, Issue 6, p17;

Building TSmiley by Nick Hodges, Issue 6, p49;

Delphi Clinic, Problem Property Editors, Issue 20, p54;

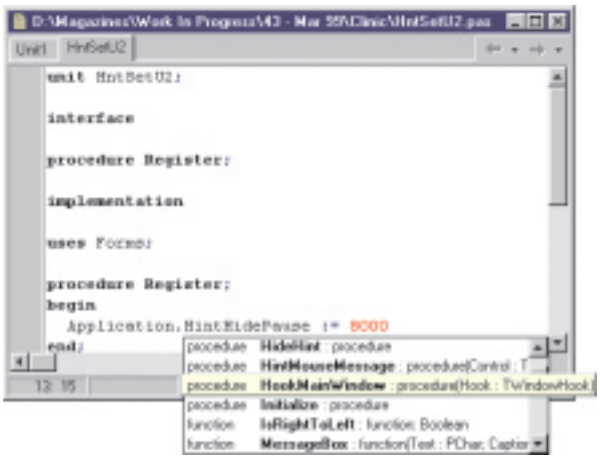
Express Yourself by Chris McNeil, Issue 23, p49;

Delphi Clinic, Alias Property Editor, Issue 30, p54.

You can also check the extensive comments in the DsgnIntf.Pas Tools API source file supplied either in Delphi's SOURCE\VCL or SOURCE\TOOLSAPI directory, depending on the Delphi version you have. To summarise its functionality and operation, when you select a component, Delphi iterates through the published properties looking for the appropriate registered property editor class for each one. Before the installation of `DataFld2`, the relevant property editor class would be `TStringProperty`. `DataFld2` supersedes this and registers `TDataField2Property` instead.

So Delphi creates an instance of `TDataField2Property`, and uses it to get the current property value (via the `GetValue` method). The value is then displayed by the Object Inspector along with all the other property values. When you select

➤ *Figure 1*



the `DataField2` property, Delphi highlights the property and draws the drop down arrow, having found that the property editor's `GetAttributes` method returns a set including `paValueList`. Delphi also manufactures a (hidden) listbox to display when the arrow is pressed. Whether this listbox has its `Sorted` property set to `True` or not is dictated by the presence (or absence) of `paSortList` in the `GetAttributes` set.

When you eventually press the arrow, Delphi calls the property editor's `GetValues` method, passing a reference to some appropriate procedure. The idea is that `GetValues` should iterate through all the available values that need to be displayed, calling this passed-in procedure for each one, which is passed the relevant value as its parameter. The implementation of this passed-in procedure presumably calls the `Add` method of the Object Inspector's hidden listbox's `Items` property.

`GetValues` manufactures a `TStrings` object and passes it to `GetValueList` to fill in. `GetValueList` uses runtime type information (RTTI) to verify that there is a published `DataSource` property in the class, and what its value is. If it is non-nil then it checks whether this data source has a data set, and if so calls its `GetFieldNames` method which takes a `TStrings` object.

➤ *Listing 4*

```
uses
  DsgnIntf, TypInfo;
type
  TDataField2Property = class(TStringProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    procedure GetValueList(List: TStrings);
    procedure GetValues(Proc: TGetStrProc); override;
  end;
function TDataField2Property.GetAttributes: TPropertyAttributes;
begin
  Result := [paValueList, paSortList, paMultiSelect];
end;
procedure TDataField2Property.GetValueList(List: TStrings);
var
  Instance: TPersistent;
  PropInfo: PPropInfo;
  DataSource: TDataSource;
begin
  Instance := GetComponent(0);
  PropInfo := TypInfo.GetPropInfo(Instance.ClassInfo, 'DataSource');
  if (PropInfo <> nil) and (PropInfo^.PropType^.Kind = tkClass) then begin
    DataSource := TObject(GetOrdProp(Instance, PropInfo)) as TDataSource;
    if (DataSource <> nil) and (DataSource.DataSet <> nil) then
      DataSource.DataSet.GetFieldNames(List);
  end;
end;
procedure TDataField2Property.GetValues(Proc: TGetStrProc);
var
  I: Integer;
  Values: TStrings;
begin
  Values := TStringList.Create;
  try
    GetValueList(Values);
    for I := 0 to Values.Count - 1 do Proc(Values[I]);
  finally
    Values.Free;
  end;
end;
procedure Register;
begin
  RegisterPropertyEditor(TypeInfo(String), TComponent,
    'DataField2', TDataField2Property)
end;
```

A throwaway sample component TPretendDataAwareControl is also supplied in a unit PretendU.pas to show the property editor working (Listing 5 shows the entire unit). Install this component (it appears on the Clinic page of the component palette by default), and place an instance of it onto a form. You can see that it has a DataSource property, along with DataField and DataField2 properties. Add a data source and a table onto the form, and set the properties of these components as shown in Listing 6.

Now go to the DataField2 property and observe the drop-down arrow present on the Object Inspector. Push the arrow button, and a list of fields will appear thanks to the code in the newly registered property editor (see Figure 2). The code in these units works in Delphi 1, 2, 3 and 4.

```
unit PretendU;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Mask, DBCtrls, DB;
type
  TPretendDataAwareControl = class(TComponent)
  private
    FDataField, FDataField2: String;
    FDataSource: TDataSource;
  published
    property DataField:  String read FDataField  write FDataField;
    property DataField2: String read FDataField2 write FDataField2;
    property DataSource: TDataSource read FDataSource write FDataSource;
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('Clinic', [TPretendDataAwareControl]);
end;
end.
```

➤ *Above: Listing 5*

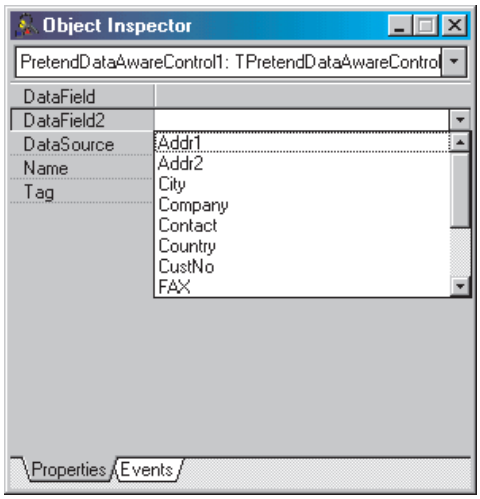
➤ *Below: Listing 6*

```
object Table1: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'CUSTOMER.DB'
end
object DataSource1: TDataSource
  DataSet = Table1
end
object PretendDataAwareControl1: TPretendDataAwareControl
  DataSource = DataSource1
end
```

Delphi 4 Disservice

Q I need to dynamically start/stop a driver service under NT 4.0 from my app. According to the help file I need two objects: a TServiceApplication and one or more TService objects. It claims that the TService object(s) will not work correctly with a 'normal' TApplication. It subsequently states that you create such an app by choosing File | New... and then selecting a Service Application from the New page. But no such item exists!

I'm using Delphi 4 Professional with Service Pack 1 on an NT 4.0 system. If there's no way to do it with File | New..., then is there a way to do it manually (with code or changing the resource file or something)?



➤ *Figure 2*

A The TService and TServiceApplication classes exist in the Professional version in the SVCMGR unit, but the IDE wizards do not. This poses a problem. The idea is to use the Application object from the SVCMGR unit instead of the one from the FORMS unit. This new Application object is of type TServiceApplication. It manages various TService objects. In the IDE a TService looks much like a data module or web module, a form designer capable of taking non-visual components only, and with properties fit for the job of representing a Windows NT service.

Now whilst you could create the TService objects under program control, clearly the wizard makes things a whole lot easier.

Here's a message picked up from the Usenet, posted some while ago by a resourceful Delphi 4 user called Jeff Overcash that tells you an easy way to deal with the problem:

'Pro has the TService and TServiceApplication classes included. *The Developer's Guide*, chapter 3, has the only documentation on them. Only the Starting Wizards were left out. I wrote a Pro version of the Wizards a month ago. You can find it at www.jgsoftware.com/nt.htm'

Thanks go to John Atkins who posted this message on CIX last August.

Missing DBGrid Events

Q Consider a program with a TDBGrid component. During execution, any column of the DBGrid can be resized by clicking and dragging the mouse at the edge of any fixed cell (the very top row). I need to know when this happens. Is there a window message that is sent to the owner form during this action?

A Well, yes, sort of. To perform the resize operation, the user clicks the left mouse button down, moves the mouse and then releases the mouse button. There are dedicated messages for all these actions. However, clearly, the resize only happens when the mouse is clicked on (or very near)

the vertical lines separating the column header cells. Rather than painstakingly calculating whether the mouse is clicked in an appropriate place we ought to take advantage of the fact that the grid already does that. So long as we can identify what the grid does to indicate a resize is in progress, we should be able to get the desired effect.

It seems that TCustomGrid, an ancestor of TDBGrid and also of TStringGrid, has a protected data field called FGridState that is given a value of gsColSizing when a column resize operation is taking place. With this information, we can write some routines that will be triggered by the arrival of the appropriate mouse-related Windows messages, and proceed from there. Instead of writing Windows message handling methods for wm_LButtonDown, wm_MouseMove and wm_LButtonUp, we will instead override some virtual TControl methods: MouseDown, MouseMove and MouseUp.

I have written a derivative of TDBGrid (in NewDBGrid.Pas) that has extra code to trigger two new events, OnResizing and OnResized. These events give you the TColumn object that is being resized, and the current width that they are being resized to. Since Delphi 1 did not support TColumn objects, the implication is that this component will not work in Delphi 1. However, the offered solution is more restrictive than that.

You might notice that you can click exactly on a title cell division to facilitate resizing, but the grid is also quite flexible in allowing you some slack to the left and right. It seems that you can click about 3 pixels to the left or right of this division, and the grid still works out what you are doing. As you move your mouse around the title cells, the cursor will indicate if a resize operation is allowed. If you resize a column to a width of 2 pixels, the logic is not impaired at all.

Some dedicated routines are employed inside the component to enable this. In order to ensure that these new events are passed information regarding the correct column, the new component should take advantage of these routines whenever the mouse is clicked at the beginning, and released at the end, of a resize operation. The problem here is these routines were private until Delphi 4. Because of extra functionality that was added by Inprise R&D for Delphi 4, they have now become protected methods.

So in fact, to make my life easy (because these routines are rather involved in their implementation), my solution currently only works in Delphi 4. The class that offers the required functionality looks like Listing 7.

Notice that when the mouse is clicked on the grid MouseDown lets the TDBGrid code do what it normally does, potentially changing the internal FGridState to gsColSizing. If this happens, we need to identify which column is being resized, and (now available) CalcDrawInfo and CalcSizingState are called to pull out the appropriate column index number.

The grid can optionally have an indicator column, indicating which is the current record, and whether the underlying dataset is in browse, edit or insert mode. This is dictated by the presence (or absence) of dgIndicator in the Options set property. If an indicator column exists, the column index we found will be one too

➤ *Listing 7*

```
TResizeEvent = procedure(Sender: TCustomDBGrid; Column:
  TColumn; Width: Cardinal) of object;
TNewDBGrid = class(TDBGrid)
private
  //Column being resized
  FColumn: TColumn;
  //Horizontal position of mouse when resize started
  FOldX: Integer;
  //Private fields for event handlers
  FOnResized,
  FOnResizing: TResizeEvent;
protected
  //Generic resize event code
  procedure DoResize(NewX: Integer; ResizeEvent:
    TResizeEvent); virtual;
  //Routines to trap resize operations
  procedure MouseDown(Button: TMouseButton; Shift:
    TShiftState; X, Y: Integer); override;
  procedure MouseMove(Shift: TShiftState; X, Y: Integer);
    override;
  procedure MouseUp(Button: TMouseButton; Shift:
    TShiftState; X, Y: Integer); override;
published
  //New events
  property OnResizing: TResizeEvent read FOnResizing
    write FOnResizing;
  property OnResized:  TResizeEvent read FOnResized
    write FOnResized;
end;
procedure TNewDBGrid.DoResize(NewX: Integer;
  ResizeEvent: TResizeEvent);
var Width: Integer;
begin
  if (FGridState = gsColSizing) and Assigned(FColumn) and
    Assigned(ResizeEvent) then begin
    //Calculate new column width
    Width := FColumn.Width + NewX - FOldX;
    //Deal with silly resize requests
    if Width <= 1 then
      Width := FColumn.Width;
    //Trigger specified event handler
    ResizeEvent(Self, FColumn, Width)
  end
end;
procedure TNewDBGrid.MouseDown(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  ColumnIndex: Integer;
  Pos, Ofs: Integer;
  State: TGridState;
  DrawInfo: TGridDrawInfo;
begin
  inherited;
  if FGridState = gsColSizing then begin
    //Set up TGridDrawInfo record for next statement
    CalcDrawInfo(DrawInfo);
    //Ask grid which cell is being resized
    CalcSizingState(X, Y, State, ColumnIndex, Pos, Ofs,
      DrawInfo);
    //Take indicator column into account
    ColumnIndex := RawToDataColumn(ColumnIndex);
    if ColumnIndex >= 0 then
      FColumn := Columns[ColumnIndex];
    FOldX := X
  end else
    FColumn := nil
end;
procedure TNewDBGrid.MouseMove(
  Shift: TShiftState; X, Y: Integer);
begin
  inherited;
  //Possible resizing occurring
  DoResize(X, FOnResizing)
end;
procedure TNewDBGrid.MouseUp(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  //Possible resize completed
  DoResize(X, FOnResized);
  //Finished resizing the column, so clear it
  FColumn := nil;
  inherited;
end;
```

```
procedure TForm1.DBGrid1Resizing(Sender: TCustomDBGrid; Column: TColumn;
  Width: Cardinal);
begin
  if Assigned(Column) then
    Caption := Format('Field %s is being resized to %d pixels',
      [Column.FieldName, Width])
end;
procedure TForm1.DBGrid1Resized(Sender: TCustomDBGrid; Column: TColumn;
  Width: Cardinal);
begin
  if Assigned(Column) then
    Caption := Format('Field %s has been resized to %d pixels',
      [Column.FieldName, Width]);
  //Give clear indication that the resize has finished
  Color := Random($1000000)
end;
```

➤ *Listing 8*

large (with respect to the `TColumn` objects), so we call `RawToDataColumn` to subtract one if this is necessary.

Assuming we get a field-related column, we can then use the `Columns` property to get the `TColumn` in question. Additionally, the current X co-ordinate is stored so that we can later identify how far the user is growing or shrinking the column.

As the mouse is moved, and as the mouse button is released, a resize operation may be happening or may be completed. Since the code to deal with these situations is similar, it is wrapped up in a common `DoResize` method. This checks that the grid is in a resizing state, that a column is indeed being resized and that the appropriate event handler exists. Assuming all these criteria are met, the new column size is calculated and the event handler is triggered.

Some sample event handlers might look like those in Listing 8. The sample project on the disk, NewGridTest.Dpr, uses this component (which will need installing) and implements these event handlers. It also has a couple of checkboxes that allow you to exclude and include both an indicator column and vertical column lines, to ensure the code works.

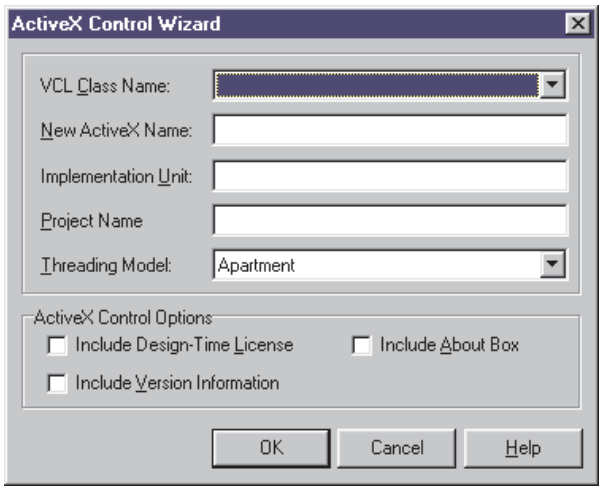
One Step ActiveX

Q Delphi 4's help on the issue of making ActiveX controls does not mention the need of a component to originate from a specific class (such as `TCustomControl`). I read in Tom Swan's *Delphi 4 Bible* that 'any Delphi component can easily be converted into an ActiveX control.' This was good news to me, as this suggests non-visual components as well as visual components. What needs to be done so that non-visual VCL components will appear in the 'eligible to change to ActiveX' list?

A To manufacture an ActiveX control in Delphi 4, you choose `File | New... | ActiveX | ActiveX Control`. This gives the dialog shown in Figure 3 which is used to generate a VCL class that represents an ActiveX. The code made use of to make the ActiveX work is called the DAX (Delphi ActiveX) Framework.

The `VCL Class Name:` combobox in Figure 3 offers all the classes that Delphi 4 thinks are acceptable for turning into ActiveX controls. This includes most of the currently installed components that are inherited from `TWinControl`, although some are excluded. The exclusions primarily relate to those which have a reliance on other components. This excludes a whole raft of components such as data-aware components (which link to data source components), and QuickReport components.

`TPageControl` is also missing. This is due to its requirement to accept VCL components as child components. If you turned a `TPageControl` into an ActiveX, other developers using other development tools would wish to add non-VCL objects to it, and this would not work. Delphi components that their authors thought should not be used for ActiveX controls are passed to `RegisterNonActiveX` during their registration.



➤ *Figure 3*

However, the exclusion rule is not exclusive. For example, `TUpDown` is in the list. The VCL `TUpDown` class has an `Associate` property that allows you to connect the `TUpDown` to another windowed control. When Delphi turns it into ActiveX, the `Associate` property is not surfaced.

So, as far as I can see, the Delphi 4 book seems to be incorrect in its assertion. To try and check, I tried setting up an ActiveX control manually, to make an ActiveX for a `TTimer`. I started by making an ActiveX for a normal control, and then went through substituting the VCL class name with `TTimer`. Unfortunately I failed, due to the requirements for a `TWinControl`. One statement of the generated source typecasts the underlying VCL component instance into a `TWinControl`, which fails with a `TTimer`.