HOW TO PUT COMPONENTS INTO A GRID

This article and the accompanying code shows how to put just about any component into a cell on a grid. By component I mean anything from a simple combobox to a more complicated dialog box. The techniques described below to anything that is termed a visual component. If you can put it into a form you can probably put it into a grid.

This was, and still may, to be submitted for publication in one of the Delphi fish wraps, but due to an overwhelming interest in the subject (lots of e-mails), I've decided to get it out there as quickly as possible. You may read this and use the techniques described here under the one condition that you read the self-serving advertisement at the end of the article.

There are no new ideas here, in fact, the basic technique simply mimics what the DBGrid does internally. The idea is to float a control over the grid. Inside DBGrid is a TDBEdit that moves around the grid. It's that TDBEdit that you key you data into. The rest of the unfocused cells are really just pictures. What you will learn here, is how to float any type of visual control/component around the grid.

That may sound complicated but it's really very simple so just follow along and you to can amaze your friends with the powerful and fun technique. <bg>

Oh yes, what I describe below is implemented in the included sample project, GRIDPROJ.DPR. If you learn better by reading code then skip the following tutorial and just read the self-serving ad at the end. Otherwise this will step though building the mentioned project.

OK, lets get started.

COMPONENT #1 - TDBLOOKUPCOMBO

First, there is a file in this download called GRIDDATA.DB which you need for this demo. The easiest thing to do is copy it to the directory that your DBDEMOS alias points to. This is the same directory which has files like Customer.DB and Orders.DB

Now you need a form with a DBGrid in it. So start an new project and drop a DBGrid into the main form.

Next drop in a TTable and set it's Alias to DBDEMOS, TableName to GRIDDATA.DB and set the Active property to True. Drop in a DataSource and set it's DataSet property to point to Table1. Go back to the grid and point it's DataSource property to DataSource1. The data from GRIDDATA.DB should appear in your grid..

The first control we are going to put into the grid is a TDBLookupCombo so we need a second table for the lookup. Drop a second TTable into the form. Set it's Alias also to DBDEMOS, TableName to CUSTOMER.DB and Active to True. Drop in a second data source and set its DataSet to Table2.

Now go get a TDBLookupCombo from the Data Controls pallet and drop it any where on the form, it doesn't matter where since it will usually be invisible or floating over the grid. Set the LookuoCombo's properties as follows.

DataSource	DataSource1
DataField	CustNo
LookupSource	DataSource2

LookupField CustNo
LookupDisplay CustNo {you can change it to Company later but keep it custno for now)

So far it's been nothing but boring point and click. Now let's do some coding.

The first thing you need to do is make sure that DBLookupCombo you put into the form is invisible when you run the app. So select Form1 into Object Inspector goto the Events tab and double click on the onCreate event. You should now have the shell for the onCreate event displayed on your screen.

```
procedure TForm1.FormCreate(Sender: TObject);
begin

end;
```

Set the LookupCombo's visible property to False as follows.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 DBLookupCombo1.Visible := False;
end;
```

Those of you who are paying attention are probably asking why I didn't just set this in the Object Inspector for the component. Actually, you could have. Personally, I like to initialize properties that change at run time in the code. I set static properties that don't change as the program runs in the object inspector. I think it makes the code easier to read.

Now we to be able to move this control around the grid. Specifically we want it to automatically appear as you either cursor or click into the column labeled DBLookupCombo. This involves defining two events for the grid, OnDrawDataCell and OnColExit. First lets do OnDrawDataCell. Double click on the grid's OnDrawDataCell event in the Object Inspector and fill in the code as follows.

```
procedure TForm1.DBGrid1DrawDataCell(Sender: TObject; const Rect: TRect;
 Field: TField; State: TGridDrawState);
begin
 if (gdFocused in State) then
 begin
    if (Field.FieldName = DBLookupCombo1.DataField) then
    begin
      DBLookupCombo1.Left := Rect.Left + DBGrid1.Left;
      DBLookupCombo1.Top := Rect.Top + DBGrid1.top;
      DBLookupCombo1.Width := Rect.Right - Rect.Left;
     { DBLookupCombo1.Height := Rect.Bottom - Rect.Top; }
      DBLookupCombo1.Visible := True;
    end;
 end;
end;
```

We the reasons for the excessive use begin/end will become clear later in the demo. The code is saying that if the State parameter is gdFocused then this particular cell is the one highlighted in the grid. Further if it's the highlighted cell and the cell has the same field name as the lookup combo's datafield then we need to move the LookupCombo over that cell and make it visible. Notice that the position is determined relative to the form not to just the grid. So, for example, the left side of LookupCombo needs to be the offset of the grid (DBGrid1.Left) into the form plus the offset of the cell into the grid (Rect.Left).

Also notice that the Height of the LookupCombo has been commented out above. The reason is that the LookupCombo has a minimum height. You just can't make it any smaller. That minimum height is larger than the height of the cell. If you un-commented the height line above. Your code would change it and then Delphi would immediately change it right back. It causes an annoying screen flash so don't fight it. Let the LookupCombo be a little larger than the cell. It looks a little funny but it works. BTW I have a solution to this but you will need to read the self-serving ad at the end to find out about it.

Now just for fun run the program. Correct all you missing semi-colons etc. Once its running try moving the cursor around the grid. Pretty cool, hu? Not! We're only part of the way there. We need to hide the LookupCombo when we leave the column. So define the grid's onColExit. It should look like this;

```
procedure TForm1.DBGrid1ColExit(Sender: TObject);
begin
 If DBGrid1.SelectedField.FieldName = DBLookupCombo1.DataField then
   DBLookupCombo1.Visible := false;
end;
```

This uses the TDBGrids SelectedField property to match up the FieldName associated with the cell with that of the LookupCombo. The code says, "If the cell you are leaving was in the DBLookupCombo column then make it invisible".

Now run it again. Was that worth the effort or what?

Now things look right but we're still missing one thing. Try typing a new customer number into one of the LookupCombo. The problem is that the keystrokes are going to the grid, not to the LookupCombo. To fix this we need to define a onKeyPress event for the grid. It goes like this;

```
procedure TForm1.DBGrid1KeyPress(Sender: TObject; var Key: Char);
begin
 if (key <> chr(9)) then
 begin
   if (DBGrid1.SelectedField.FieldName = DBLookupCombo1.DataField) then
   begin
     DBLookupCombo1.SetFocus;
     SendMessage(DBLookupCombo1.Handle, WM_Char, word(Key), 0);
   end;
 end;
end;
```

This code is saying that if the key pressed is not a tab key (Chr(9)) and the current field in the grid is the LookupCombo then set the focus to the LookupCombo and then pass the keystroke over to the LookupCombo. OK so I had to use a WIN API function. You don't really need to know how it works just that it works.

But let me explain a bit anyway. To make Window's SendMessage function work you must give it the handle of the component you want to send the message to. Use the component's Handle property. Next it wants to know what the message is. In this case it's Window's message WM_CHAR which says I'm sending the LookupCombo a character. Finally, you need to tell it which character, so word(Key). That’s a typecast to type word of the events Key parameter. Clear as mud, right? All you really need to know is to replace the DBLookupCombo1 in the call to the name of the component your putting into the grid. If you want more info on SendMessage do a search in Delphi's on-line help.

Now run it again and try typing. It works! Play with it a bit and see how the tab key gets you out of "edit mode" back into "move the cell cursor around mode".

Now go back to the Object Inspector for the DBLookupCombo component and change the LookupDIsplay property to Company. Run it. Imagine the possibilities.

COMPONENT #2 - TDBCOMBO

I'm not going to discuss installing the second component, a DBCombo, because I don't really have anything new to say. It's really the same as #1. Here's the incrementally developed code for your review.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 DBLookupCombo1.Visible := False;
 DBComboBox1.Visible := False;
end;

procedure TForm1.DBGrid1DrawDataCell(Sender: TObject; const Rect: TRect;
 Field: TField; State: TGridDrawState);
begin
 if (gdFocused in State) then
 begin
   if (Field.FieldName = DBLookupCombo1.DataField) then
   begin
     DBLookupCombo1.Left := Rect.Left + DBGrid1.Left;
     DBLookupCombo1.Top := Rect.Top + DBGrid1.top;
     DBLookupCombo1.Width := Rect.Right - Rect.Left;
{      DBLookupCombo1.Height := Rect.Bottom - Rect.Top; }
     DBLookupCombo1.Visible := True;
   end
   else if (Field.FieldName = DBComboBox1.DataField) then
   begin
     DBComboBox1.Left := Rect.Left + DBGrid1.Left;
     DBComboBox1.Top := Rect.Top + DBGrid1.top;
     DBComboBox1.Width := Rect.Right - Rect.Left;
{     DBComboBox1.Height := Rect.Bottom - Rect.Top; }
     DBComboBox1.Visible := True;
   end
 end;

end;

procedure TForm1.DBGrid1ColExit(Sender: TObject);
begin
 If DBGrid1.SelectedField.FieldName = DBLookupCombo1.DataField then
   DBLookupCombo1.Visible := false
 else If DBGrid1.SelectedField.FieldName = DBComboBox1.DataField then
   DBComboBox1.Visible := false;
end;

procedure TForm1.DBGrid1KeyPress(Sender: TObject; var Key: Char);
begin
 if (key <> chr(9)) then
 begin
   if (DBGrid1.SelectedField.FieldName = DBLookupCombo1.DataField) then
```

```
  begin
    DBLookupCombo1.SetFocus;
    SendMessage(DBLookupCombo1.Handle, WM_Char, word(Key), 0);
  end
  else if (DBGrid1.SelectedField.FieldName = DBComboBox1.DataField) then
  begin
    DBComboBox1.SetFocus;
    SendMessage(DBComboBox1.Handle, WM_Char, word(Key), 0);
  end;
 end;
end;
```

COMPONENT #3 - TDBCHECKBOX

The DBCheckBox gets even more interesting. In this case it seems appropriate to leave something in the non-focused checkbox cells to indicate that there's a check box there. You can either draw the "stay behind" image of the checkbox or you can blast in a picture of the checkbox. I chose to do the later. I created two BMP files one that's a picture of the box checked (TRUE.BMP) and one that's a picture of the box unchecked (FALSE.BMP). Both of these BMP files are included with this download. Put two TImage components on the from called ImageTrue and ImageFalse and attach the BMP files to there respective Picture properties. Oh yes you also need to put a DBCheckbox component on the form. Wire it to the CheckBox field in DataSource1 and set the Color property to clWindow.

First edit the onCreate so it reads as follows;

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 DBLookupCombo1.Visible := False;
 DBCheckBox1.Visible := False;
 DBComboBox1.Visible := False;
 ImageTrue.Visible := False;
 ImageFalse.Visible := False;
end;
```

Now we need to modify the onDrawDataCell to do something with cells that do not have the focus. Here comes the code.

```
procedure TForm1.DBGrid1DrawDataCell(Sender: TObject; const Rect: TRect;
 Field: TField; State: TGridDrawState);
begin
 if (gdFocused in State) then
 begin
   if (Field.FieldName = DBLookupCombo1.DataField) then
   begin
         ...SEE ABOVE
   end
   else if (Field.FieldName = DBCheckBox1.DataField) then
   begin
     DBCheckBox1.Left := Rect.Left + DBGrid1.Left + 1;
     DBCheckBox1.Top := Rect.Top + DBGrid1.top + 1;
     DBCheckBox1.Width := Rect.Right - Rect.Left{ - 1};
     DBCheckBox1.Height := Rect.Bottom - Rect.Top{ - 1};
     DBCheckBox1.Visible := True;
   end
   else if (Field.FieldName = DBComboBox1.DataField) then
```

```
   begin
          ...SEE ABOVE
   end
 end
 else {in this else area draw any stay behind bit maps}
 begin
  if (Field.FieldName = DBCheckBox1.DataField) then
  begin
    if TableGridDataCheckBox.AsBoolean then
      DBGrid1.Canvas.Draw(Rect.Left,Rect.Top, ImageTrue.Picture.Bitmap)
    else
      DBGrid1.Canvas.Draw(Rect.Left,Rect.Top, ImageFalse.Picture.Bitmap)
  end
 end;
```

It's the very last part we're most interested in. If the state is not gdFocused and the column in CheckBox then this last bit executes. All it does is check the value of the data in the field and if it's true it shows the TRUE.BMP otherwise it shows the FALSE.BMP.

I created the bit maps so they are indented so you can tell the difference between a focused and unfocused cell.

Make onColExit look like this;

```
procedure TForm1.DBGrid1ColExit(Sender: TObject);
begin
 If DBGrid1.SelectedField.FieldName = DBLookupCombo1.DataField then
   DBLookupCombo1.Visible := false
 else If DBGrid1.SelectedField.FieldName = DBCheckBox1.DataField then
   DBCheckBox1.Visible := false
 else If DBGrid1.SelectedField.FieldName = DBComboBox1.DataField then
   DBComboBox1.Visible := false;
end;
```

Edit onKeyPress to;

```
procedure TForm1.DBGrid1KeyPress(Sender: TObject; var Key: Char);
begin
 if (key <> chr(9)) then
 begin
   if (DBGrid1.SelectedField.FieldName = DBLookupCombo1.DataField) then
   begin
     DBLookupCombo1.SetFocus;
     SendMessage(DBLookupCombo1.Handle, WM_Char, word(Key), 0);
   end
   else if (DBGrid1.SelectedField.FieldName = DBCheckBox1.DataField) then
   begin
     DBCheckBox1.SetFocus;
     SendMessage(DBCheckBox1.Handle, WM_Char, word(Key), 0);
   end
   else if (DBGrid1.SelectedField.FieldName = DBComboBox1.DataField) then
   begin
     DBComboBox1.SetFocus;
     SendMessage(DBComboBox1.Handle, WM_Char, word(Key), 0);
   end;
 end;
```

```
end;
```

Finally, here's the last trick. The caption of the checkbox needs to change as the user checks or unchecks the box. My first thought was to do this in the TDBCheckBox's onChange event, the only problem is that it doesn't have one. So I had to go back to the Windows API and send another message. "SendMessage(DBCheckBox1.Handle, BM_GetCheck, 0, 0)" which returns a 0 if the box is unchecked, otherwise it's checked.

```
procedure TForm1.DBCheckBox1Click(Sender: TObject);
begin
 if SendMessage(DBCheckBox1.Handle, BM_GetCheck, 0, 0) = 0 then
    DBCheckBox1.Caption := '  ' + 'False'
 else
    DBCheckBox1.Caption := '  ' + 'True'
end;
```

That's it. Hopefully you learned something. I've tried this technique with dialog boxes. It works and it's simple. Have fun with it. You don't really need to completely understand it as long as you know how to edit the code and replace the above component names with with the name of the component you want to drop into the grid.

Here's the self serving advertisement.

Self Serving Advertisement
THE DBLOOKUPCOMBOPLUS COMPONENT

The original DBLookupCombo that ships with Delphi is a very powerful component but wait till you see **DBLookupComboPlus** it's got all those extra features you wished you had in the original. By the way, one of it's features is that you can size it to fit into a grid cell correctly.

The new **DBLookUpComboPlus** gives you all the lookup power of Delphi's original DBLookUp-Combo plus much, much more;

Major Enhancements
* The ability to sort the drop down list.
* The ability to search through the dropdown list via an incremental keyboard search. (Quicken Style).
* Autofill - The text is automatically filled in as the user types characters. (Quicken Style).
* The ability to enter new records into the lookup table on the fly. (OnNewLookUpRec)
* The control can stay in a Read/Write style even when the Data source and Lookup Source are different. (this goes with the previous point)
* A new event to support the filling the lookup list with the results from a TQuery(OnPrepareList).
* Ad new event that allows the lookup to return multiple data elements instead of just the one specified by the LookupField property. (OnGridSelect)
* The drop down list can be either left or right justified.
* The drop down list can be changed to a rise up list.
* The BoarderStyle property is available.
* The ability to hide or show the drop down speed button.
* New event to simplify the populating of multiple main table fields from the lookup table.

Other minor enhancements over the original control include;
- Corrects the two memory leaks found in TDBLookUpCombo.
- Automatic placement of dropdown list to fit the screen has been enhanced.
- If the old TDBLookUpCombo was placed on a form with fsStayOnTop set then the list would drop down behind the form and could not be seen. This problem has been corrected in

TDBLookupComboPlus.
- The minimum height of the edit box portion of the control has been corrected to be more inline with the behavior of TEdit, TDBEdit, TCombo, etc.

You get all that with no compromise in performance. It's fast. It's efficient.

Down load the full featured control from the Delphi Database or 3rd Party Library in the Delphi Forum. It's shareware. The cost, - $20 for the component and only another $10 for the source code (if you already own the VCL source).

If you have any question please drop me, Alec Bergamini, at 75664,1224. My company is Out & About Productions.

Out & About Productions is a band of dedicated professionals specializing in RAD with Delphi and custom component development. We are also the local no-it-alls on C/S development with Oracle, BTrieve, Netware SQL.(or what ever they are calling it today).

We through out all our VB and Power Builder manuals a few months ago so don't even call it that's what you want.

FAX : (619) 566-0210
VOICE : Sorry, only given to customers.