The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?
Just email Brian Long, our Delphi Clinic Editor, on 76004.3437@compuserve.com or write/fax us at The Delphi Magazine

Changing The Defaults

Q How do I change the default size of the Object Inspector and Edit Window, set a new default form font and change the default project directory?

A The default form font and default editor size can be specified in the DELPHI.INI file in the Windows directory as specified in \DELPHI\DOC\INIFILE.TXT. To set the default font to the Windows 3.1 dialog default, ensure the following entry occurs in the `FormDesign` section:

```
[FormDesign]
DefaultFont=MS Sans Serif, 8, bold
```

To set the editor to a new size, add this entry to the `Editor` section:

```
[Editor]
DefaultWidth=500
DefaultHeight=400
```

To set accurate values, you can use WinSight (from the Delphi Program Manager group). Manually set the editor to your desired size, then choose WinSight's `Spy | Find Window` and click on the caption bar of the Delphi editor. This will cause WinSight to highlight the editor details, including its co-ordinates, from which you can calculate the width and height.

To set a new default size and position for the Object Inspector, you need to make Delphi generate a default desktop file (\DELPHI\BIN\DELPHI.DSK) with appropriate sizing information in it. Follow these instructions.

1. Choose `File | Close Project` (if you have a project open).
2. Choose `Options | Environment` and look on the `Preferences` tab.
3. If `Autosave options: Desktop` is not checked then check it. Also ensure the `Desktop contents: Desktop only` radio button is selected.
4. Set your desired Object Inspector size and choose `File | Exit`.
5. Restart Delphi. If you checked `Autosave options: Desktop` in step 3, uncheck it again.

To change the default directory that Delphi offers to save projects in, you must change its idea of its own current directory. This can be achieved by changing the Windows 3.1 Program Manager properties for the Delphi icon, or the Windows 95 shortcut properties. This is not an ideal solution, since during one session Delphi can have its current directory changed, but it's the best I can come up with at this time.

TDBLookupList & TDBLookupCombo Exceptions

Q The `TDBLookupList` and `TDBLookupCombo` components' `LookupSource` data source property must be connected to a `TTable` component. Why is this?

A The lookup components use the table's `SetKey` and `GotoKey` methods to look up the `LookupField` value. A query does not have these methods. Avoiding the problem requires using a third-party component that does its lookups in some other way. *[Take a look at Steve Troxell's 'Surviving Client/Server' column from Issue 5 for some hints. Editor]*

Comment On Issue 6 Article

Q I understood that writing low level things like system interrupt handlers as you demonstrated in Issue 6 (*Callbacks In Windows And The BDE: Part 3*) required rather more than just writing code. Don't you have to set up segment attributes?

A That's right – the article omitted these details (oops). Any code that may get called from an interrupt handler needs to be present in memory when it is needed. SysUtils itself ensures that it is in memory by using the following compiler directive to set up the segment attributes:

```
{$C MOVEABLE PRELOAD PERMANENT}
```

This directive should really also be present in my INTRUPTU.PAS and INTRUPU2.PAS source files, as some of their code may be called from the SysUtils interrupt handler. In truth there is still more that can be done to make things safer. Windows resources also have attributes to dictate whether they remain available in memory, or get brought in as required. Forms are stored as resources, and the form in INTRUPU2.DFM is used from the interrupt handler. If you have Resource Workshop, you can load the DFM file up, select `TACTIONFORM` and `Resource | Memory options...`, and uncheck `Load on call` and `Discardable`.

Making Field Objects Without The Fields Editor

Q We wish to standardise access to our tables for all our programmers. This means we will create a `TCustomerTable` component (with default `TableName` and `DatabaseName` properties, etc), which will be used in preference to the standard `TTable` component. Is it possible to programmatically create field objects in this new

component's constructor which will be available to the programmer at design time through the Object Inspector? These field objects would be just like those made by the Fields Editor, but without having to explicitly use it. If this is the case, can we also manufacture calculated fields in this way?

To do this will simply require a scan through the source code of the Fields Editor to see what it does. The Fields Editor (internally known as the Dataset Designer) is in the DELPHI\LIB directory in DSDESIGN.DFM and DSDESIGN.DCU and... oh dear – no source code! This means we'll have to work it out for ourselves...

When the Fields Editor is used, it is something external to the `TDataset` derivative it is working on (I'll just refer to `TTables` for the purposes of this discussion). It manufactures a new object which gets stored in the form file, but is then connected to the appropriate table. Once it has done this, the new field object is around forever, or until it or the table is deleted. Each time the form is opened at design or run time, the field object will be brought back to life. If we make a field object from inside a new table object we need to be careful to match this operation. It needs to be created only when the component is dropped onto the form, but not when a form is being opened at design or run time.

In principle, all we need to do to get the field object in the Object Inspector is create it, specifying it is owned by the form (not the table), and set its properties, including its name (we'll stick with the Fields Editor convention of the table component name concatenated with the field name). Setting the properties is the tricky part: we need to make sure that `Size`, `FieldName`, `DataType` and of course the class type are all correct before finishing, otherwise the field object will not be accepted, and the table component will fail to construct. We could decide to force the component writer to hardcode all these properties, but this is unnecessary. We should be able to glean the information from the table itself – remember that a table component has a `FieldByName` method which returns a field object when given a field name (providing the table is open, we'll open it if it isn't).

This is fine if we just want one field object. Listing 1 shows a component (TABLE.PAS) which represents a customer table and makes a field object for the `Company` field.

However, for more than one field the plan falls down. You may know that when using the Fields Editor, once you add even one field object, the table view is restricted to just that one field. `FieldByName` will fail for any other fields and so we won't have access to the properties we want to set up. To remedy this the second version of the component, TABLE2.PAS in Listing 2, uses another `TTable`, set up to point at the same table, to access the normal run-time field objects. This is done in `MakeFields`, which takes an array of field names as a parameter.

As for calculated fields, we need to supply more information: the field name, data type and size (or zero if it is not relevant). We still need to ensure that an object of an appropriate type is constructed (`TStringField`, `TBooleanField` etc) but this time we cannot rely on existing run-time objects to copy. Instead, I have used a `TFieldDef` object in the `MakeCalculatedField` method, which does it all for me.

The last issue is the calculation of the calculated fields. The object sets up its own `OnCalcFields` handler, called `OldOnCalcFields`, which calculates the value of the `Taxable` field (these calculated values show up at design time if the table's `Active` property is `True`). It could have been just one line, but there is something else to consider. The new table component will still have `OnCalcFields` showing on the Object Inspector's Events page. If a user double-clicks on this (or presses `Ctrl-Enter`) a replacement

➤ *Listing 1*

```
unit Table;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, DB, DBTables;
type
  TNewTable = class(TTable)
  public
    constructor Create(AOwner: TComponent); override;
  end;
procedure Register;

implementation

constructor TNewTable.Create(AOwner: TComponent);
var
  Field, OldField: TField;
  SaveActive: Boolean;
begin
  inherited Create(AOwner);
  DatabaseName := 'DBDEMOS';
  TableName := 'CUSTOMER';
  { Only make objects when asked by user not when
    reading in from form stream }
  if not (csLoading in AOwner.ComponentState) then begin
    SaveActive := Active;
    Active := True;
    { Specify target field here }
    OldField := FieldByName('Company');
    Field := TFieldClass(OldField.ClassType).Create(Owner);
    Field.FieldName := OldField.FieldName;
    Field.Name := Name + Field.FieldName;
    Field.Size := OldField.Size;
    if not SaveActive then
      Active := False;
    Field.DataSet := Self;
  end;
end;

procedure Register;
begin
  RegisterComponents('Samples', [TNewTable]);
end;
end.
```

handler will be made, losing the ready-made calculations. This may or may not be desirable so I have allowed for either.

The object adds a replacement `OnCalcFields` event property, which has one additional parameter: `DoDefault`. If a user makes a new `OnCalcFields` handler, the value of `DoDefault` (which defaults to `True`, but can be changed in the handler) dictates whether the original calculations will take place or not. An example project showing the second component in action is provided as TBLTEST.DPR. At design time the calculated field is given values, at run-time `DoDefault` is set to `False` and so no values are given.

Delphi Hasn't Got The Power

Q If Delphi is such an advanced language, why is it such a pain to raise a number to a power? In Visual Basic we can say `Z = X^Y` but the best we can do in Delphi is

```
Z := Exp(Ln(X) * Y);
```

A Rather than enter into a battle of which language is the more capable, I'll simply say that Delphi 2.0 remedies this, but for 16-bit development here is a more complete implementation of a power function. Unlike the short expression above, this caters for powers of zero and negative numbers. Listing 3 is the routine (note the use of the often overlooked functions `Frac` and `Odd`) taken from the POWERU unit on the disk, which is used by a project called POWTEST.DPR. The project has two edit boxes for entering the number and the power to raise it to, and also a label to display the result. The power is calculated by tabbing from one edit box to another (using their `OnExit` event handlers) or by pressing the `Enter` key after typing in a new number (`OnKeyPress` event handlers).

One or two things in the listing are worth mentioning here before leaving the subject. If a non-integral power is requested of a negative number, this is detected and an error is signalled by way of an exception being raised manually. This is done rather than perform the mathematical operation which would yield the problem for a good reason. DLLs written in Delphi

➤ *Listing 2*

```
unit Table2;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, DB, DBTables;
type
  TCalcFieldsEvent =
    procedure (DataSet: TDataSet; var DoDefault: Boolean)
      of object;
  TNewTable2 = class(TTable)
  private
    FOnCalcFields: TCalcFieldsEvent;
    procedure OldOnCalcFields(DataSet: TDataSet);
  public
    constructor Create(AOwner: TComponent); override;
    procedure MakeFields(const FieldNames: array of String);
    procedure MakeCalculatedField(const FieldName: String;
      DataType: TFieldType; Size: Word);
  published
    property OnCalcFields: TCalcFieldsEvent
      read FOnCalcFields write FOnCalcFields;
  end;
procedure Register;
implementation
constructor TNewTable2.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  DatabaseName := 'DBDEMOS';
  TableName := 'CUSTOMER';
  MakeFields(['Company', 'CustNo', 'TaxRate']);
  MakeCalculatedField('Taxable', ftBoolean, 0);
  inherited OnCalcFields := OldOnCalcFields;
end;
procedure TNewTable2.MakeFields(
  const FieldNames: array of String);
var
  CopyTable: TTable;
  Field, CopyField: TField;
  Loop: Byte;
begin
  { Only make objects when asked by user
    not when reading in from form stream }
  if not (csLoading in Owner.ComponentState) then begin
    { Make normal table object (it will have all fields
      available. If we were to rely on this table, after the
      first field object is added, we wouldn't see any other
      fields) }
    CopyTable := TTable.Create(nil);
    try
      { Set up copy table properties and open it }
      CopyTable.DatabaseName := DatabaseName;
      CopyTable.TableName := TableName;
      CopyTable.Open;
      { Loop for each new field object }
      for Loop := Low(FieldNames) to High(FieldNames) do
        begin
        { Find the normal run-time field }
        CopyField := CopyTable.FieldByName(FieldNames[Loop]);
        { Construct a new object of the appropriate class
          type. This is the thing which will end up in the
          Object Inspector }
        Field :=
          TFieldClass(CopyField.ClassType).Create(Owner);
        { Tie it to a field }
        Field.FieldName := FieldNames[Loop];
        { Give the object a name }
        Field.Name := Name + FieldNames[Loop];
        { Set the size up correctly }
        Field.Size := CopyField.Size;
        { Insert the field in this table }
        Field.DataSet := Self;
      end;
    finally
      CopyTable.Free; { Finished with the copy table now }
    end;
  end;
end;
procedure TNewTable2.MakeCalculatedField(const FieldName:
  String; DataType: TFieldType; Size: Word);
var
  Field: TField;
begin
  { Only make objects when asked by user
    not when reading in from form stream }
  if not (csLoading in Owner.ComponentState) then
    { Use a field definition object to save code }
    with TFieldDef.Create(nil, FieldName, DataType,
      Size, False, 0) do
      try
        { Make appropriate field object }
        Field := CreateField(Owner);
        Field.Calculated := True;
        { Sort its name out, so it will appear
          in the Object Inspector }
        Field.Name := Name;
        { Insert it into the table's field list }
        Field.DataSet := Self;
      finally
        Free;
      end;
end;
procedure TNewTable2.OldOnCalcFields(DataSet: TDataSet);
var
  DoDefault: Boolean;
begin
  DoDefault := True;
  if Assigned(FOnCalcFields) then
    FOnCalcFields(DataSet, DoDefault);
  if DoDefault then begin
    { Calculated field calculation needs to be placed here }
    FieldByName('Taxable').AsBoolean :=
      FieldByName('TaxRate').AsFloat > 0;
  end;
end;
procedure Register;
begin
  RegisterComponents('Samples', [TNewTable2]);
end;
end.
```

have a responsibility to trap any software exceptions that may happen (an important and understated point), to prevent them being left with the EXE (which may not have been written in Delphi). If this function were to be called in a DLL it would therefore be desirable to trap the exception and deal with it. However, the DLL would not get a chance. Math errors and other hardware exceptions such as GPFs are picked up directly by the EXE which, if it is written in Delphi, will then turn the problem into a Delphi software exception. To allow the DLL a look-in, the code generates the software exception directly, which can be trapped, thereby avoiding the issue.

The two edit controls share an `OnExit` event handler called `CalculatePower`, which calls the `Power` function with the values gleaned from the edit controls. They also share an `OnKeyPress` handler which detects the `Enter` key, and calls the `CalculatePower` handler.

```
function Power(Number, Exponent: Extended): Extended;
const
  SInvalidOp = 65428; { from SysUtils.Inc }
begin
  if Number > 0 then
    { +ve number, any power: x^y = exp(ln(x)*y) }
    Result := Exp(Ln(Number) * Exponent)
  else if Exponent = 0 then
    { Any number to power zero: x^0 = 1 }
    Result := 1
  else
    { -ve number, any power - only valid for whole powers}
    if (Frac(Exponent) = 0) and (Exponent <= MaxInt) then begin
      Result := Exp(Ln(-Number) * Exponent);
      { If the power is odd, result is -ve }
      if Odd(Trunc(Exponent)) then Result := Result * -1;
    end else
      { Fractional power of negative number not allowed so give exception }
      raise EInvalidOp.CreateRes(SInvalidOp);
end;
```

➤ *Listing 3*

```
var OldMode: Word;
...
OldMode :=  SetErrorMode(sem_FailCriticalErrors + sem_NoOpenFileErrorBox);
AssignFile(F, c:\delphi\readme.txt);
try
  Reset(F);
except
  on EInOutError do
    {Handle problem};
end;
SetErrorMode(OldMode);
```

➤ *Listing 4*

Windows File Errors

Q When I try and open a file that is already opened in another program, Windows puts up a system modal error telling me it is already in use and asks me if I wish to cancel the operation or try again. If I cancel, I then get a normal Delphi I/O *File Access Denied* exception. Is there a way to prevent this Windows error box?

A There are at least two ways of stopping the problem; the first is to modify the `FileMode` variable before calling `Reset` or `Rewrite` (see my *File Handling* article in this issue for more details on these symbols):

```
FileMode := fmOpenReadWrite +
  fmShareDenyNone;
```

The other approach is to explicitly tell Windows not to show its error box using the `SetErrorMode` API, as shown in Listing 4. Using file sharing symbols tends to be the preferred way.

Toggling Shift Key States

Q I read your *Typecasting Explained: Part 2* article in the November issue which, among other things, showed one way to access the BIOS Data Area in conventional memory. I tried to use an approach like this to toggle the state of `Caps Lock` and `Num Lock`, as I can successfully do under DOS, but to no avail.

A You can forget about talking to the BIOS Data Area for this chore: you need to use Windows APIs. Windows maintains an array of bytes representing the state of all the keys on the keyboard (indexed by their virtual key codes). For the `Caps Lock`, `Num Lock` and `Scroll Lock` keys, the low bit is important since it acts as a toggle. Switch the bit to the other setting and the state of the that key changes. In other words, to change the state, you flip the bit (which can be done with an `xor` operation). A project called TOGGLE.DPR is included on the disk showing the old approach and also the correct approach. Here is the code which toggles the `Caps Lock` key state:

```
var Keyboard: TKeyboardState;
procedure TForm1.CapsLockClick(
  Sender: TObject);
begin
  GetKeyboardState(Keyboard);
  Keyboard[vk_Capital] :=
    Keyboard[vk_Capital] xor 1;
  SetKeyboardState(Keyboard);
end;
```