The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

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

Auto-Repeat Toolbuttons

Q You showed us in Issue 22 how to make `TButton` components auto-repeat. This allowed the user to keep the left mouse button held down on a button and get repeated clicks, a bit like the auto-repeat behaviour of the keyboard. I have been trying to get the same behaviour with tool buttons but have not got anywhere with it yet. Can you help?

A The article in Issue 22 looked at auto-repeat buttons because the `TDBNavigator` component supports auto-repeat on a couple of its speed buttons, and some people had asked me how to get the same effect with normal buttons.

To achieve the same result with any kind of button requires a similar approach, just taking care to test it thoroughly. The project AutoRptBtns.dpr on the disk has a tool button, speed button and normal button demonstrating this auto-repeat behaviour. The tool button just beeps the PC speaker each time its `OnClick` event is triggered, the speed button writes the time on the form's caption, and the normal button gives the form a random colour. To get auto-repeat clicking for these buttons, a number of steps have been taken.

Firstly, the buttons share the same `OnMouseDown`, `OnMouseMove` and `OnMouseUp` event handlers, which do most of the work. Secondly, a timer has been added to the form, with its `Enabled` property set to `False`. Finally, to allow the code to be shared among a number of controls, a private `TControl` data field called `Ctrl` has been defined in the form class. The code is in Listing 1.

The general idea is that, when the mouse is clicked down on one of these buttons, the timer is enabled and has its `Interval` set to some appropriate value. Each time the timer ticks, the button's `Click` method is called in order to trigger the `OnClick` event handler.

You can see that, to overcome the fact that `Click` is protected in the base `TControl` class (the type of the `Ctrl` identifier), I have used a simple access class. This is quite a common trick to access protected members of a class: define a shallow descendant of the class in the unit where you are working, then typecast the object into that access class. You instantly get access to the protected members.

If the mouse is moved whilst the left mouse button is still down, the

➤ *Listing 1: Auto-repeat buttons.*

```
type
  TForm1 = class(TForm)
    ToolBar1: TToolBar;
    ToolButton1: TToolButton;
    Timer1: TTimer;
    Button1: TButton;
    SpeedButton1: TSpeedButton;
    procedure AutoRepeatMouseDown(Sender: TObject; Button:
      TMouseButton; Shift: TShiftState; X, Y: Integer);
    procedure AutoRepeatMouseUp(Sender: TObject; Button:
      TMouseButton; Shift: TShiftState; X, Y: Integer);
    procedure AutoRepeatMouseMove(Sender: TObject; Shift:
      TShiftState; X, Y: Integer);
    procedure Timer1Timer(Sender: TObject);
    procedure ToolButton1Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure SpeedButton1Click(Sender: TObject);
  private
    Ctrl: TControl;
  end;
...
const
  { pause before repeat timer starts (ms) }
  InitRepeatPause = 400;
  { pause before successive hits (ms)}
  RepeatPause     = 200;
type
  //Access class needed to get at Click method,
  //which is protected in the TControl base class.
  //TControl is used so this code will work against
  //any visual control with an OnClick event handler
  TControlAccess = class(TControl);
procedure TForm1.AutoRepeatMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  Ctrl := Sender as TControl;
  Timer1.Interval := InitRepeatPause;
  Timer1.Enabled := True;
  //Don't let the normal click happen when mouse is
  //released, just the faked clicks from the timer will
  //suffice
  Ctrl.ControlState := Ctrl.ControlState - [csClicked];
end;

procedure TForm1.AutoRepeatMouseUp(Sender: TObject; Button:
  TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  //If timer didn't tick at all, do one click
  if Timer1.Enabled
    and (Timer1.Interval = InitRepeatPause) then
    TControlAccess(Ctrl).Click;
  Timer1.Enabled := False;
  Ctrl := nil;
end;
procedure TForm1.AutoRepeatMouseMove(Sender: TObject; Shift:
  TShiftState; X, Y: Integer);
begin
  if Assigned(Ctrl) and (csLButtonDown in Ctrl.ControlState)
    then with Ctrl do
      Timer1.Enabled :=
        PtInRect(Rect(0, 0, Width, Height), Point(X, Y))
end;
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Timer1.Interval := RepeatPause;
  try
    TControlAccess(Ctrl).Click;
    //If button has been disabled as a result of what
    //happens in its OnClick method, shutdown timer
    Timer1.Enabled := Ctrl.Enabled;
  except
    Timer1.Enabled := False;
    raise;
  end;
end;
procedure TForm1.ToolButton1Click(Sender: TObject);
begin
  Beep
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  Color := Random($1000000);
end;
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  Caption := TimeToStr(Time)
end;
```

timer will be disabled if the mouse moves off the button and re-enabled when it moves back on. This helps the usability of this auto-repeat feature.

When the mouse is released, the timer is disabled; however, if this happens before the timer has had a chance to tick once, meaning the button's `OnClick` event handler has not yet been called, then this is taken care of by calling the `Click` method at this point.

One final mouse issue is that, by default, when the mouse is released the `OnClick` event will trigger just by the default nature of the button. To prevent this extra, probably unwanted, click the `OnMouseDown` event handler tweaks the button's `ControlState` set property to tell it that it is not in the process of being clicked at all.

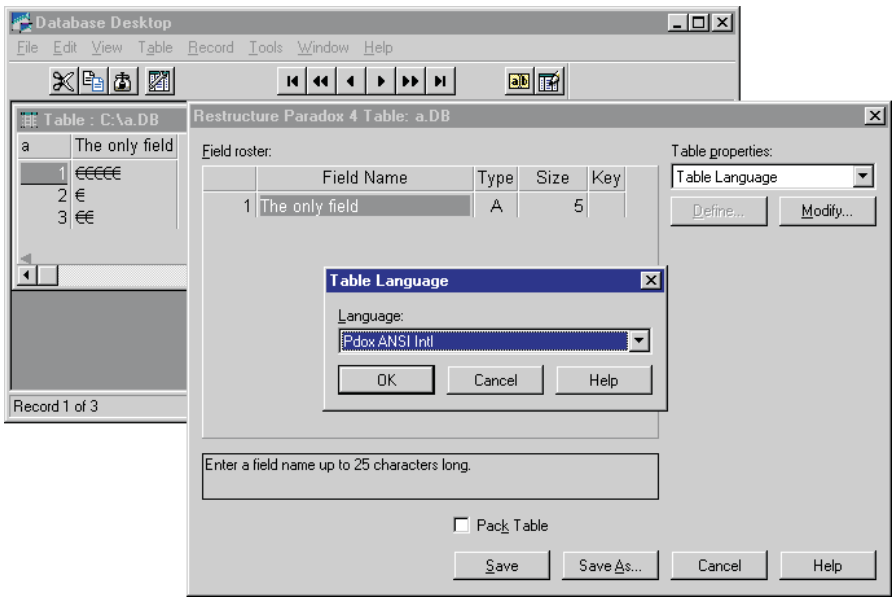
The timer has two intervals set during this process. When the mouse is first clicked on the button, its `Interval` is set to an initial delay defined in a constant called `InitRepeatPause` (400 ms). Each time the timer ticks, however, `Interval` is changed to `RepeatPause` (a smaller value of 200 ms).

Paradox And The Euro symbol

Q I am developing an international application and I need to save the Euro currency symbol ( ) in a field in a Paradox table. When I save it and then recall the record, the Euro symbol changes to a hash sign (#) or empty square, or some other character. To support international characters, the Table Language was set in Database Desktop to `Paradox (intl)`.

A The `Paradox (intl)` language driver was designed for use in the DOS environment primarily. It makes use of the characters set up in DOS code pages and defines a case-insensitive sort order compatible with the old Paradox for DOS international sort order. Unfortunately, Microsoft has not added the Euro symbol to the DOS code pages, so the language driver does not support it.

To verify this, I performed a test with the Database Desktop running on Windows 98. I made a new Paradox table with one character field and set the Table Language to `Paradox (intl)`. Then I added a record to the table, inserted the Euro symbol and tried to save the record. For my trouble, I was presented with the error *Character(s) not supported by Table Language*.



➤ *Figure 1: Setting the right language driver for the Euro symbol.*

The Euro symbol was added to some of the Windows fonts (Arial, Courier New, Tahoma and Times New Roman) as ANSI character 128, although other fonts still have an empty square in position 128. This means that, in an application that is using one of these fonts (such as a word processor), you should be able to get the Euro symbol by ensuring `Num Lock` is on, holding down the `Alt` key and typing `0128` on the numeric key pad. Note that it has to be a four digit number to get an ANSI character. If you hold `Alt` and enter the three digit number `128`, you will get the ASCII character `128`, which is Ç.

Since the Euro is an ANSI character, you should get better results by setting the Table Language to `Paradox ANSI Intl`. A quick test in Database Desktop reinforces this suggestion (see Figure 1).

Lost Variables In The Debugger

Q When debugging my application, I sometimes try to evaluate a local variable and am told it is unavailable due to optimisation. What does this mean, and how can I overcome it?

A When Delphi is installed, it sets up a default number of compiler switches. One of the switches that is enabled is optimisation (`Project | Options... | Compiler | Code generation | Optimization`). This asks the compiler to generate more efficient code to help your program execute more efficiently, using a number of rules and strategies that it has been programmed with.

In general, local variables are implemented by the entry code of their subroutine allocating some space on the program stack for them. This space is synonymous with the local variable for the extent of the subroutine's execution and is freed by the routine's exit code. This means that the local variable is truly in existence throughout the routine's life, and so has some value at all times (even if it is a garbage value due to the variable not having been assigned).

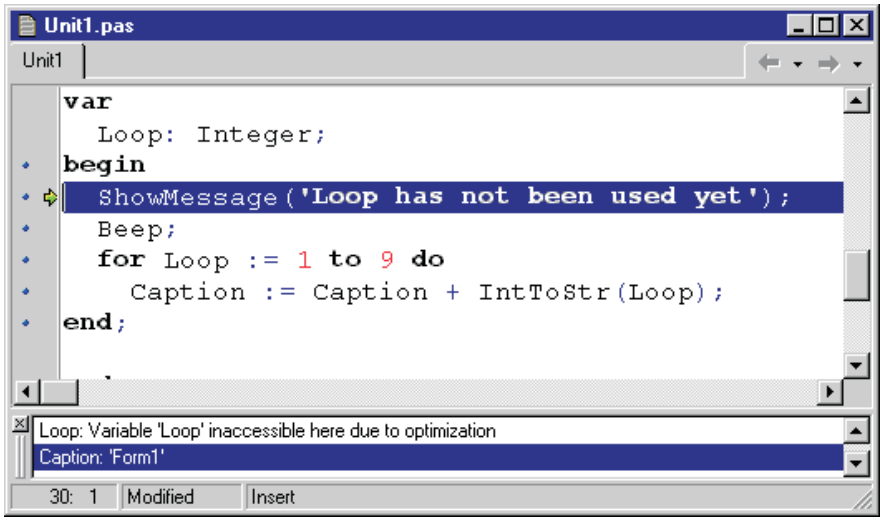
The program stack resides in normal RAM. It is a known fact that reading a value from RAM is slower than reading a value from a CPU register. One of the optimisations

that the compiler performs (when the switch is on) is to store local variables in CPU registers where possible (bearing in mind CPU registers are only 32 bits wide) rather than on the stack. However, since there are only a handful of CPU registers, this has to be done with care.

The compiler analyses the use of the variable in your code and typically associates the register with the variable only when it is being used. At all other times in the routine, the register is used for other tasks (maybe other local variables, or temporary working space for calculations, etc). So if you try and examine the value of a variable at a point before it is used or after it has been used, the debugger will alert you to the fact that it doesn't really exist at that point: it has been optimised away (see Figure 2).

The remedy for this, which helps make debugging easier, is to turn the optimisation switch off. Whilst developing an application, I tend to turn optimisation off, and turn stack frames, range checking and overflow checking on. In fact, since these are the settings I use whilst developing any project, I make these the default project settings (using the `Default` checkbox at the bottom of the project options dialog as shown in Figure 3). 32-bit Delphi stores default project options in defproj.dof in the Bin directory, whereas Kylix stores them in the file defproj.kof in the ~/.borland directory (the .borland hidden directory under your home directory).

When the development cycle is over, the runtime checks can be turned off, as can stack frames, and optimisations can be turned on. You should perform some testing to ensure that things still work fine with these different compiler settings, but ultimately you will get smaller and faster code for deployment with these different compiler settings.



➤ *Figure 2: The Loop variable has been optimised away.*

```
TCar = class
public
  Speed: Integer;
end;
...
var
  Car: TCar;
...
if Car.Speed <> 0 then
  Car.Speed := 999;
```

➤ *Listing 2: Using a public data field.*

Public Properties

Q What's the point of declaring a property as public? After all, a published property will be shown in the Object Inspector and a protected property can be made published in a descendant class.

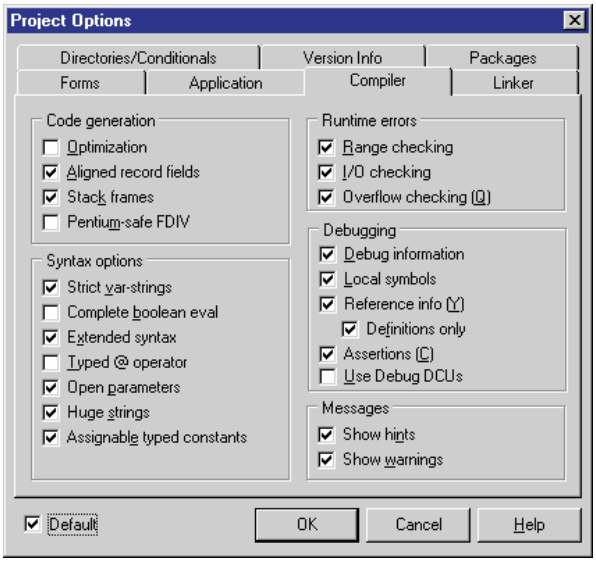
A The answer to this question lies in knowing what a property is and what it does. The question appears to be formed on the basis of thinking that a property's usefulness lies solely in being able to appear on the Object Inspector, but this is only part of it. Let's take a brief refresher course on what a property is.

Firstly, let's look at a few scenarios which do not involve properties. Take a simple class that represents a car where the speed can be changed. A simple implementation is shown in Listing 2. This type of implementation is normally frowned upon, as the class has no control over the public data field. Any code in the program can modify the car's speed at will, potentially assigning ludicrous values to it.

Consequently, when data is defined in a class, we are advised to make it private and use data accessor routines (see Listing 3), so called *data hiding*. C++ programmers are very familiar with these, as they use them all the time. The beauty of accessor routines (sometimes called *getters* and *setters*) is that you can add extra code in them to do additional tasks, such as validation, as shown in the listing. Users of the class are obliged to call these routines in order to access the data, and thereby invoke your extra code.

The downside to using getters and setters is that the code that

➤ *Figure 3: Setting default compiler options.*



calls them can end up looking a bit messy. Ultimately we are dealing with the speed of the car (one data field), but we have to call two routines to manage it.

The property concept was introduced to simplify this messy coding, whilst maintaining the data hiding that we have so far. Instead of forcing the programmer to deal with two identifiers for one value, a single property is defined to do the job. When the property is read from, the getter is called, when the property is assigned a value, and that value is passed as a parameter to the setter. The result is shown in Listing 4.

As you can see, we are back to using one identifier (the property), but we still have the data hiding and validation thanks to the getter and setter. You can think of a property giving us *implementation hiding*: it hides the implementation of our data hiding mechanisms. Incidentally, whether the getter and setter are made private or protected, and whether they are marked as virtual or not, are choices made by the implementer.

Whilst to the casual observer a property may appear to be a data item, Listing 4 shows that it is not. A property has no storage of its own. A property is defined in terms of a data type (so the compiler can validate its use in expressions) and what happens when it is read from and written to.

➤ *Listing 4: A property in use.*

```
TCar = class
private
  FSpeed: Integer;
  function GetSpeed: Integer;
  procedure SetSpeed(Value: Integer);
public
  property Speed: Integer read GetSpeed write SetSpeed;
end;
...
var
  Car: TCar;
...
if Car.Speed <> 0 then
  Car.Speed := 999; //Car's speed will be set to 100
```

➤ *Listing 5: An optimised version of the property.*

```
TCar = class
private
  FSpeed: Integer;
  procedure SetSpeed(Value: Integer);
public
  property Speed: Integer read FSpeed write SetSpeed;
end;
```

```
TCar = class
private
  FSpeed: Integer;
public
  function GetSpeed: Integer;
  procedure SetSpeed(Value: Integer);
end;
...
function TCar.GetSpeed: Integer;
begin
  Result := FSpeed
end;
procedure TCar.SetSpeed(Value: Integer);
const
  TopSpeed = 100;
begin
  //Validate value
  if (Value <> FSpeed) then begin
    if Value > TopSpeed then
      Value := TopSpeed;
    FSpeed := Value
  end
end;
...
var
  Car: TCar;
...
if Car.GetSpeed <> 0 then
  Car.SetSpeed(999); //Car's speed will be set to 100
```

➤ *Listing 3: Using data accessor routines.*

The property syntax is quite flexible, as you can see if you look it up in the Help. Notice that in our case the getter does nothing other than return the data field. Well, since that is all the getter does, the property syntax allows us to dispense with it completely, leaving us with Listing 5. The syntax of a property allows a read operation to either call a getter to retrieve a value, or directly return the value of a (typically private) data field. Similarly, the syntax also allows a write operation to either pass the value to a setter, or directly update the value of a data field.

Because the property retrieves a value, either from a data field or through a function call, and can be given a value to store in a data field or pass to a method, it can be manipulated in code quite like it is in fact a data item. In fact, this was the whole design behind the property syntax: to have an item that looks like a piece of data but can do other things behind the scenes. Data with a side-effect, if you like.

Just to help reinforce this idea of a property being more than a piece of data, think about the `Color` property of a form. If you change `Color` to a different value, the form actually repaints itself with a different colour. If `Color` was just a data item, assigning a new value to it would cause that value to be stored in the relevant area of memory and nothing more would happen. The fact that the form will redraw tells us that code executes when you assign to the `Color` property (code in the `Color` property's setter routine).

All this so far has been an overview of properties. Getting back to the question, I should say that when a component author defines a property that can be used by a programmer, they have the choice of making it public or published. It is important to realise that in either case, the property will be accessible at runtime through code. However, if the property is published, it will also appear on the Object Inspector when the component is selected on the Form Designer.

The reason someone chooses either public or published is really a case of which one makes sense. If there is a case for pre-setting the value of the property in the design-time environment, then the property will be published. If it does not make sense to set it at design-time, the property will be public.

The `TListBox` component gives a good example of this. It makes sense sometimes to pre-fill the listbox with a number of text strings, so the `Items` property is published. However, at design-time it is not possible to select any items in a listbox (that is something which happens at runtime only), so the `ItemIndex` property is public. At runtime you can still access either property via a listbox object, but at design-time only the published property is accessible for pre-setting with a value.

The other point raised in the question regards protected component properties. It doesn't really affect the answer given so far, but since it was mentioned, I should address it. When component authors build new components, they often do it in two levels. Let's consider a fictitious component `TFoo`. The author will build all the functionality into a base class called `TCustomFoo`, including all the properties. However, all the properties that *could* validly appear on the Object Inspector are placed in the protected section of the class (but public properties remain public). `TFoo` then inherits from `TCustomFoo`, adding nothing to the class, but it publishes all the properties that it wishes to appear on the Object Inspector.

The reason this is done is to allow scope for other `TFoo`-like components which do not need all the properties visible at design-time. A good example of this would be `TCustomEdit`, which has no published properties, but has a lot of protected and public properties. The `TEdit` descendant adds no functionality but publishes all the protected properties from `TCustomEdit` as well as some from other classes in the inheritance hierarchy, such as `Text`. However, another descendant of `TCustomEdit`, `TDBEdit`, does add extra code, but elects to not publish the `Text` property.

```
TCustomFoo = class(TComponent)
private
  FSomeProp: Integer;
  FSomeOtherProp: Integer;
  procedure SetSomeProp(Value: Integer);
  procedure SetSomeOtherProp(Value: Integer);
protected
  property SomeProp: Integer read FSomeProp write SetSomeProp;
  property SomeOtherProp: Integer read FSomeOtherProp write SetSomeOtherProp;
end;
TFoo = class(TCustomFoo)
published
  property SomeProp;
  property SomeOtherProp;
end;
TDifferentFoo = class(TCustomFoo)
published
  property SomeProp;
  //Note that this component will not have SomeOtherProp
  //available on the Object Inspector
end;
```

➤ *Listing 6: Publishing protected properties.*

Without `TCustomEdit`, implementing `TDBEdit` would be a problem due to the occurrence of an unwanted `Text` property, accessible at design-time and runtime. As it is, the `Text` property is protected in `TDBEdit` and cannot be accessed by programmers.

Listing 6 shows the idea with the `TFoo` component hierarchy. `TCustomFoo` defines two properties which could appear on the Object Inspector, but are protected. `TFoo` publishes them both, but `TDifferentFoo` doesn't want both of them to appear and so leaves `SomeOtherProp` being protected.

Popup Data Grids

Q I heard someone say it is possible to have a master/detail query in a MIDAS client/server application that makes use of popup data grids. I've never seen this done. Can you enlighten me?

A This was a feature added in Delphi 4 and offers an alternative to having two grids on the MIDAS client form: the detail grid can appear as a grid in a new floating window. To see one in action, we need to set up a MIDAS server with a master/detail relationship on it and then set up the client application to show it.

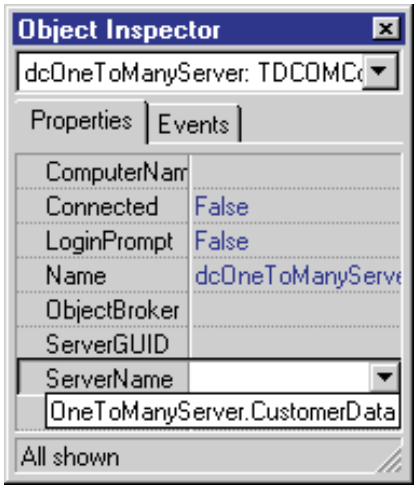
Given that MIDAS keeps changing in each version of Delphi, the steps that follow are for MIDAS 3 in Delphi 5. They may also work in Delphi 4, but I have a feeling there may be the odd difference that you need to take into account here and there.

```
object tblCustomer: TTable
  DatabaseName = 'DBDEMOS'
  TableName = 'customer.db'
end
object tblOrders: TTable
  DatabaseName = 'DBDEMOS'
  IndexName = 'CustNo'
  MasterFields = 'CustNo'
  MasterSource = dsCustomer
  TableName = 'orders.db'
end
object dsCustomer: TDataSource
  DataSet = tblCustomer
end
```

➤ *Listing 7: A master/detail relationship.*

Firstly, the server which, for simplicity of building, will talk to the client via (D)COM. Make a fresh project (the sample server on the disk is OneToManyServer.dpr) and add a remote data module to it (`File | New... | Multitier | Remote Data Module`). When it asks for a CoClass name, specify `CustomerData`, which means the remote data module will be accessible from the client with the `ProgID` of `OneToManyServer.CustomerData`.

On the data module, set up two datasets to represent tables which have a master/detail relationship; for example, two `TTable` components and a `TDataSource`, or two `TQuery` components and a `TDataSource`. I chose to use tables, simply because they are easier, and have mapped them onto the `Customer` and `Orders` tables from the `DBDEMOS` database. The three components are set up with names



➤ *Figure 4: Locating the MIDAS server at design-time.*

and properties as shown in Listing 7.

To make this data available through the remote data module, we next add a `TDataSetProvider` from the `MIDAS` page of the Component Palette and connect it to `tblCustomer` via the `DataSet` property. Rename the component `dspCustomer`.

In order to make the server usable it must be registered. This can be done by running it with a command-line parameter of `/regserver`, whereupon it will run, register itself and terminate. However, it is easier to simply run the program normally, where it will register itself and then need to be closed normally.

With the server complete and registered we can now create the client (an example client is on the disk as OneToManyClient.dpr). Make another project and drop a `TClientDataSet` (called `cdsCustomer`) and `TDCOMConnection` (called `dcOneToManyServer`) on the form. The DCOM connection object needs information about the server application, which we will give through its `ServerName` property.

When you drop down the list of available values, the property editor makes sure that the `ProgIDs` of any MIDAS servers are displayed (see Figure 4). This rather handy feature of identifying all the MIDAS servers on a machine is ultimately achieved by the property editor through the undocumented procedure `GetMIDASAppServerList` in the `MConnect` unit.

Having chosen the server, you can verify that it can be connected to by setting the `Connected` property to `True`. If all is well, you should see your server application appear on the screen. You may as well leave it connected, since other properties we need to set will use the connection (and re-establish it first if you close it now).

Now go to the client dataset and link it to the DCOM connection component via the `RemoteServer` property and then choose which provider from the server to map onto using the `ProviderName` property. You should see the only option offered is `dspCustomer`, the name of our dataset provider. The property editor used the connection to the server to ascertain this information.

The final steps are to add a `TDataSource` and connect it to the client dataset, then add a `TDBGrid` and connect it to the data source. Also, to avoid the server always running when the client is open at design-time, set the DCOM connection object's `Connected` property to `False`, then make an `OnCreate` event handler for the form and call the client dataset's `Open` method there.

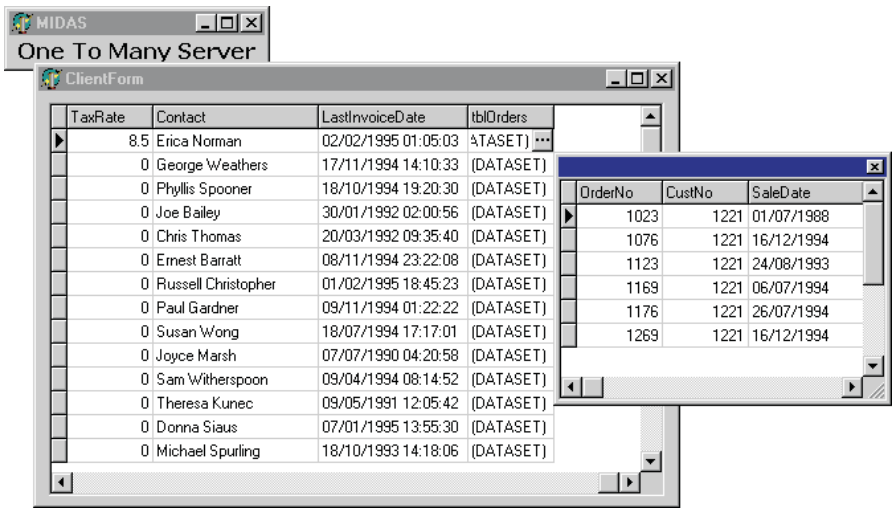
When you run the client program, the server will be launched and a connection will be made. This allows the client grid to be filled in with data. If you scroll across to the last column in the grid, it has a heading of `tblOrders` and each 'value' is `(DATASET)`. Clicking on one of these cells a couple of times will invoke the in-place editor and show an ellipsis button (...). Pressing the button displays the extra floating grid (Figure 5). As you select different rows in the main grid on the form, the floating grid updates to show the detail records for the selected customer.

If clicking a cell to select it, clicking it again to get the ellipsis button, then clicking the ellipsis button, seems too much trouble (as it seems to be for many people) there are several ways to make this more usable. First, the main grid on the form can have `dgAlwaysShowEditor` set in the `Options` property. This removes one click from the equation.

Perhaps a better way would be to remove the funny looking end column from the main grid and to float the popup grid through code. You could also use one of the ideas shown in my *Drag And Dock* article from Issue 63 to allow the grid to be displayed and hidden as needed, maybe using a menu item or checkbox in conjunction with an action. Another client project on the disk, OneToManyClient2.dpr, shows this idea.

When the form is created, the client dataset is opened as usual. Then the columns in the grid are searched for the one whose type is `TDataSetField`, which is hidden from view when found. After that, the popup grid is displayed on the screen just below the normal grid using the main grid's `ShowPopupEditor` method. Once the popup

➤ *Figure 5: A popup TDBGrid.*

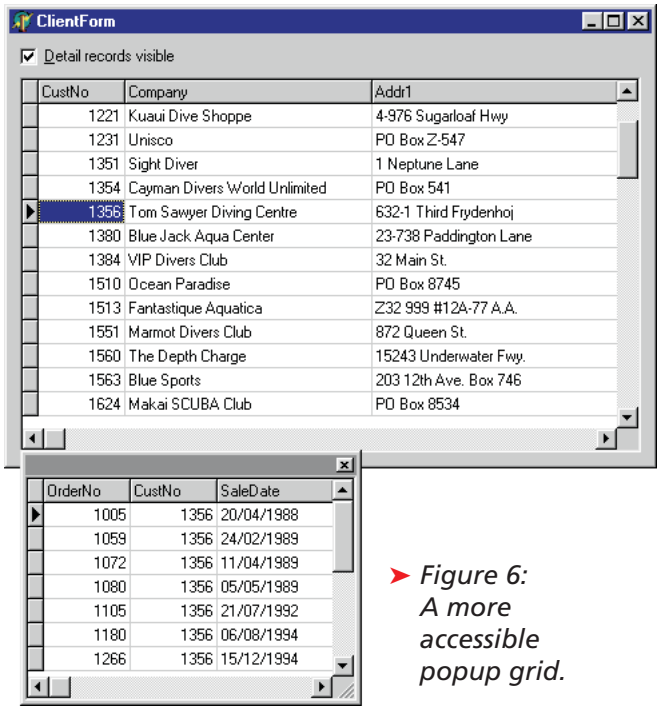


```
TClientForm = class(TForm)
private
  FPopupGrid: TCustomDBGrid;
end;
...
procedure TClientForm.FormCreate(Sender: TObject);
var
  I: Integer;
  Pt: TPoint;
  Col: TColumn;
begin
  cdsCustomer.Open;
  //Locate and hide dataset field column
  Col := nil;
  for I := 0 to DBGrid1.Columns.Count - 1 do
    if DBGrid1.Columns[I].Field is TDataSetField then begin
      Col := DBGrid1.Columns[I];
      Col.Visible := False;
      Break //There is one dataset field column
    end;
  if not Assigned(Col) then
    raise EDatabaseError.Create(
      'Trouble brewing... dataset field not found');
  //Display popup grid below other grid
  Pt := Point(DBGrid1.Left, DBGrid1.Top + DBGrid1.Height);
  Pt := ClientToScreen(Pt);
  DBGrid1.ShowPopupEditor(Col, Pt.X, Pt.Y);
  //Because the popup grid has been displayed,
  //it remains in existence with the main grid, so
  //save reference to it for later use
  for I := 0 to DBGrid1.ComponentCount - 1 do
    if DBGrid1.Components[I] is TCustomDBGrid then
      FPopupGrid := TCustomDBGrid(DBGrid1.Components[I]);
  if not Assigned(FPopupGrid) then
    raise EDatabaseError.Create(
      'Trouble brewing... popup grid not found')
end;
```

➤ *Listing 8: Setting up for a more accessible popup grid.*

```
procedure TClientForm.actShowPopupGridExecute(
  Sender: TObject);
begin
  FPopupGrid.Visible := not actShowPopupGrid.Checked
end;
procedure TClientForm.actShowPopupGridUpdate(
  Sender: TObject);
begin
  actShowPopupGrid.Checked := FPopupGrid.Visible
end;
```

➤ *Listing 9: Using an action to control the popup grid.*



➤ *Figure 6: A more accessible popup grid.*

grid is visible, the code searches for it by iterating through all the components owned by the main grid. Once found, a reference to it is stored, since the popup grid will now stay in existence until the main grid is destroyed. This code can be found in Listing 8.

There is a checkbox on the form connected to an action in an action list (`actShowPopupGrid`). The checkbox reflects and controls the visibility of the popup grid (see Figure 6). Toggling the checkbox makes the popup grid appear and disappear. Also, thanks to the action's `OnUpdate` event handler, closing the popup grid will update the checkbox (see Listing 9).

You can get a more thorough discussion of master/detail relationships and MIDAS in Bob Swart's *Under Construction* column from Issue 46. In fact, if you tire of the popup grid, you can go back to having two grids and two client datasets on the client form, as discussed in that article.

Acknowledgements

Thanks go to Borland's Steve Axtell for the information on Paradox language drivers.