The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project?

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

InterBase Generators

Q I have master and detail tables in an InterBase database and I'm using `TQuery` components to select, insert and update. The tables have a common linking unique key field and I use an InterBase generator to make new values for it when new master records are inserted.

Once I execute the query to insert a new record (and use the generator to make a new unique value) how do I read the record back? I cannot find a way to identify the new record, since the unique key was generated by InterBase itself, and so I don't know its value. This means I cannot insert detail records linked to the new master record. How do I fix this?

A For any readers who are unaware, a generator is a mechanism in InterBase for generating unique, sequential numbers. You can find information on how to create generators in the InterBase online help.

InterBase allows you to write a special query (or stored procedure) that will return the current value of any generator. It relies on operating against one of the InterBase system tables. Given a generator `X`, the following query will generate a one-record, one-field result set with the generator X's current value in it:

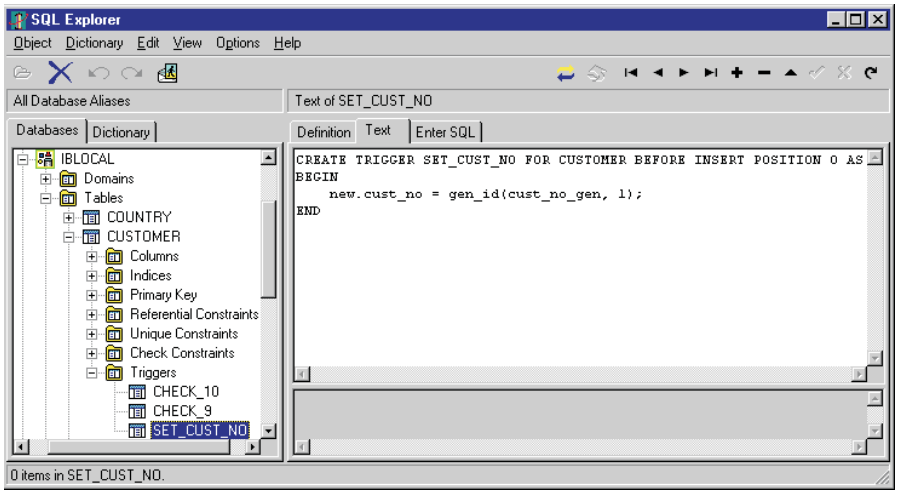
```
SELECT GEN_ID(X, 0)
  FROM RDB$DATABASE
```

Whenever you need to know the value, you can execute this query and read the first field in the first (indeed only) record. In other words, you read the `Fields[0].AsInteger` property.

Alternatively, you can substitute the 0 for a 1, and the generator will increment and return the new value. This gives the next unique number in the generator series, meaning that instead of calling `GEN_ID` in the next `INSERT` query, you can instead use a parameter which gets set to this new value.

Another possibility is the `AutoGenerateValue` property of a `TField`, which is an attempt to workaround the complexities of this without coding. This property was added in Delphi 5 and you can see how it works with a simple experiment based on the sample InterBase database, accessed via the `IBLOCAL` alias.

➤ *Figure 1: The CUSTOMER table trigger.*



The sample `CUSTOMER` table has a trigger set up, so any new inserted record gets a `CUST_NO` field value from the `CUST_NO_GEN` generator (see Figure 1). This means that if you insert a new record in the `CUSTOMER` table, you can put any garbage value you like in the `CUST_NO` field, but by the time the record is stored, it will have a correct, unique value from the generator.

The `CUSTOMER` table from this database has an associated detail table, `SALES`. GenVal.dpr is a simple project that has two `TTable` components on its form (`tblCustomer` and `tblSales`), showing the `CUSTOMER` and `SALES` tables from the database, connected in a master/detail relationship.

The `tblCustomer` component has its `AutoRefresh` property set to `True`, which is necessary for the `AutoGenerateValue` property to work. The Fields Editor is used to make persistent field objects for all fields in the table, and the `CUST_NO` field object has the `AutoGenerateValue` property set to `arAutoInc` (the field is, after all, an automatically incrementing field). The `Required` property is then set to `False` since, although the field must ultimately have a value, it will get one thanks to the trigger.

If you run the application and insert a new record in the `CUSTOMER` grid (by pressing the `Insert` key), you can enter new customer details. Ignore the `CUST_NO` field when entering data, but look what happens to it when you post a new record. You will see that it is filled in with the value that was written by the trigger (the new generator value).

The application also has a label at the bottom of the form which is updated when the program starts, and after each post operation on the `CUSTOMER` table. It runs a simple

query that follows the scheme outlined earlier, in order to report the current generator value (see Figure 2).

VCL Debugging Query

Q A quick but baffling question: since moving up to Delphi 5, I am now sent deep into the VCL code whenever I am debugging. For example, I'll try to do an `F7` (or `Shift+F7`) on one of my function calls, and suddenly I'll find myself in the `Classes`, `Controls` or `Forms` VCL unit source file.

I've removed any VCL path from

```
Project|Options...|Directories/
  Conditionals|
  Debug source path
```

to no avail. How can I set up the debugger so that, by default, I only debug my own code?

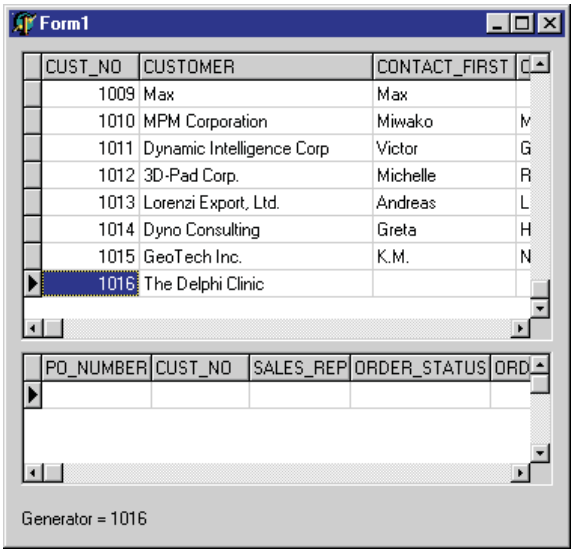
A The option you chose to modify has no effect on this particular problem. The `Debug source path` option is designed to allow the debugger to find *your* source files, in the event that you have moved them to a new directory since the last compilation.

The debugger locates the VCL and RTL source code (when it needs it) thanks to `Tools | Environment Options... Library | Browsing path`. But you should not change the paths in this entry (unless you move the VCL/RTL source). Instead, you should tell the debugger to use the non-debug versions of the pre-compiled VCL/RTL units, rather than the debug versions, which it is currently doing.

Delphi 5 has a new compiler option on the `Compiler` page of the project options dialog called `Debug DCUs`. By default it is off, meaning that the compiler will use the VCL/RTL DCUs from Delphi's Lib directory. These DCUs were compiled with no debug information within them.

When the option is on (as it is in the questioner's case), the compiler uses DCUs from Delphi's Lib\Debug directory. These were compiled with debug information, and the debugger uses the aforementioned `Browsing` path entry to locate the corresponding source files, so it can load them into the editor as required and show which source lines are being stepped through.

In short, turn `Debug DCUs` off to solve the problem.



➤ *Figure 2: The auto-refreshed field.*

BDE Paradox User Limit

Q I vaguely remember there is some kind of limitation to Paradox database performance if there are more than 20 or so instances of the same program running, writing to the same local Paradox database at the same time. However, I can't find anything on this topic in the help file. Do you know anything about it?

A The documented limits for the BDE are at:

```
http://community.borland.com/
  article/0,1410,15159,00.html
```

and include a limit of 48 BDE clients on a system and 63 Paradox sessions with tables open on a system.

None of the limits in the document seem to be as restrictive as the questioner suggests, but there is also another problem of using the BDE with CGI type web applications that would be limiting in this way. The problem is hinted at in the BDEREADME.TXT file, where it says too many BDE initialisations can yield the error: *Operation not applicable*. The file does not mention the exact limits, but you would be restricted to about 30 concurrent connections.

Strange Form Behaviour

Q I'm trying to make a sort of message form which can appear on the screen for a few seconds when the program is busy. I invoke the form with a call to its `Show` method, execute the 'busy' code, then call the form's `Hide` method. The message form appears on the screen, but the strange thing is that all the label captions are missing. Do you have an explanation?

A This situation is much like the one you face when invoking a splash screen, and has much the same solution.

Calling a form's `Show` method causes the basic form shape to be drawn on the screen, but all the details on the form will be drawn the next time a `WM_PAINT` message is processed. Unfortunately, if you follow the call to `Show` with more time-consuming code, which is then followed by code that hides the form, no message processing takes place for that form.

With a splash screen, calling `Show` and then creating all the rest of the auto-created forms, followed by hiding the splash screen would pose the same problem: no message processing takes place, so no drawing takes place on the form.

So now on to the obvious question: *Why does no message processing take place in these scenarios?* To answer the question relies on knowing how Windows applications work. Windows is a message-driven system and Windows applications operate by waiting for messages of interest to arrive so they can be processed. In fact, the

individual windows in a Windows application are waiting for specific messages of interest to arrive.

Messages arrive in one of two ways. Either they are directly handed to the target window for immediate processing or they are placed in an application message queue for processing at the next convenient moment. Keyboard, mouse and paint messages are queued and so are not processed immediately.

At the heart of any Delphi application is a *message loop* (sometimes called a *message pump*) that keeps plucking messages out of the queue, one at a time, and sending each one to the target window. Take, for example, a message that indicates a button has been pressed. This message will be removed from the queue and passed to the button, ultimately resulting in the button's `OnClick` event handler being executed (if it exists). What happens in the event handler may be negligibly short, or may take some time to execute. In the questioner's case, there is clearly some event handler with time consuming code in it.

However long the event handler takes to do its job, the message loop will be unable to pull any more messages from the queue until the event handler finishes and returns control to the message loop. In the questioner's case, an event handler is executing and calls the message form's `Show` method. This gets the form's outline drawn on the screen and causes a `WM_PAINT` message to be added to the message queue, targeted for that form.

However, before the event handler ends, more code executes to do whatever takes the time, and then the form is hidden. By the time the event handler ends and the `WM_PAINT` message is pulled for processing, the target form is no longer around.

So the problem occurs because no queued messages are automatically processed whilst an event handler executes, only between event handler executions. To fix the problem we need to know how to force messages to be processed at our command.

The traditional solution to this general problem is to call `Application.ProcessMessages`, which initiates an additional, temporary message loop, processing the pending queued messages. The downside to this approach is that it is not only paint messages which get processed, but all messages in the queue.

The question really calls for knowing how to force a drawing operation instantly, rather than waiting for the paint message to be pulled from the queue and processed. To accommodate this, all VCL controls have an `Update` method that will refresh the view of the control if a paint message is pending, otherwise it will do nothing.

Listing 1 shows the order that will make things work. Immediately calling `Update` after `Show` will make sure the form is fully displayed before the rest of the lengthy code takes place.

Again, splash screens use a similar idea. The MastApp demo (in Delphi's Demos\DB\MastApp directory) has a splash screen. If you view the project source (`Project | View Source` in Delphi 4 or later) you will see a similar scheme used.

➤ *Listing 1: Displaying a message form.*

```
MessageForm := TMessageForm.Create(Application);
try
  MessageForm.Show;
  MessageForm.Update;
  //Lengthy code goes here
  MessageForm.Hide
finally
  MessageForm.Free;
  MessageForm := nil
end;
```

Task Manager Figure

Q I found the *Optimised Working Set* response you gave in Issue 58 most interesting. It does, however, beg one question. In NT, Task Manager shows the memory used by each process. Try as I may, I cannot find a Delphi/API procedure which gives the same information so that I can monitor memory from within the program.

I have tried `GetHeapStatus` and `GlobalMemoryStatus` and cannot relate the values obtained to anything NT reports, at least not anything useful.

A It took a bit of searching, but I found some information on my MSDN CD. It comes from the November 1996 issue of *Microsoft Systems Journal* in Matt Pietrek's *Under The Hood* column. To quote Matt: 'Have you ever wondered about the *Mem Usage* column in the Windows NT 4.0? Where does it get those numbers from? Those numbers are the actual working set of each process.'

He goes on to discuss an NT-only API called `QueryWorkingSet`, which returns information on every memory page currently in use by a specified process. To calculate the working set size, you multiply the number of pages being used by a process by the size of a memory page (which is 4Kb in Win32, but rather than use a fixed value, you should get it from a call to the `GetSystemInfo` Win32 API). The working set, therefore, is the amount of memory used by a given process for everything (code, data, resources, and so on).

`QueryWorkingSet` takes a handle to the process in question, a pointer to a block of memory to fill with memory page data, and the size of the memory block. It fills the buffer with a `DWord` indicating how many memory pages are to be described, and then one `DWord` per memory page. So the important figure is in the first `DWord` in the memory buffer.

It is handy to have a general idea of how many pages will be described so you can make the memory block large enough. To do this, observe the figures obtained in Task Manager whilst your application is running, divide by 4 (memory page size) and add some value to it to allow for growth.

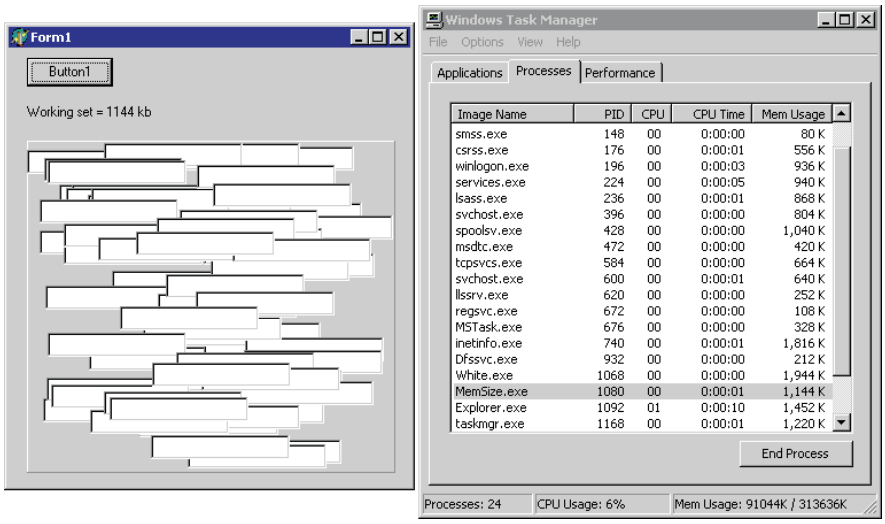
It is easy to get your own process ID (`GetCurrentProcessId`), but to get a handle to your process requires a call to `OpenProcess`. Since we are merely getting information about the process, the access mode parameter value can be `PROCESS_QUERY_INFORMATION`.

The event handler in Listing 2 (from the MemSize.dpr project on the disk) seems to do the job just fine, displaying the memory usage in kilobytes in a label (shown in Figure 3). The application has a button which creates randomly placed edit controls as a way of eating up memory, to test the accuracy of the memory measurement.

Note that Delphi 4 was the first version to have the `PSAPI` unit, containing the declaration of `QueryWorkingSet`.

```
uses
  PSAPI;
procedure TForm1.Timer1Timer(Sender: TObject);
var
  SI: TSystemInfo;
  PageSize: DWord;
  CurrentProcessHandle: THandle;
  DWords: array[0..2048] of DWord;
  Res: DWord;
const
  KiloByte = 1024;
begin
  GetSystemInfo(SI);
  PageSize := SI.dwPageSize div KiloByte;
  CurrentProcessHandle :=
    OpenProcess(PROCESS_QUERY_INFORMATION, False, GetCurrentProcessId);
  //Working set only valid on NT
  if Win32Platform = VER_PLATFORM_WIN32_NT then
    if QueryWorkingSet(CurrentProcessHandle, @DWords, SizeOf(DWords)) then
      Label1.Caption :=
        Format('Working set = %d kb', [DWords[0] * PageSize])
    else begin
      //If QueryWorkingSet fails, show error no./msg.
      Res := GetLastError;
      Label1.Caption :=
        Format('Cannot calculate working set, Win32 error %d (%s)',
          [Res, SysErrorMessage(Res)])
    end
  else
    Label1.Caption := 'Working set only valid on NT platforms'
end;
```

➤ *Listing 2: Memory usage as per Task Manager.*



➤ *Figure 3: Recreating the Task Manager's Mem Usage figure.*

Property Editor Question

Q I am writing a component and am having problems with one of its properties. The property is basically an integer (a subrange actually), but like many of the VCL component properties, a number of the values are represented by special constants. I want the user to be able to enter the constant names in the Object Inspector, and for the Object Inspector to automatically display the constant names for the appropriate values.

I have tried using the `RegisterIntegerConsts` routine in order to achieve this goal, but it appears to do nothing. The Object Inspector continues to display numbers for all values of this property.

A Unfortunately, you have chosen the wrong tool for this job. `RegisterIntegerConsts` is not designed to help the design-time support offered by the Object Inspector. Instead, it is designed to work in conjunction with the streaming system.

The VCL source code calls `RegisterIntegerConsts` for `TCursor` (in Controls.pas), `TColor` and `TFontCharSet` (in Graphics.pas). Generally speaking, you find no design-time stuff in the VCL source code. The R&D people strive to keep design-time code in separate units (which they supply only a few of, in Delphi's Source\ToolsAPI and Source\Property Editors directories). The implication, therefore, is that `RegisterIntegerConsts` is not a design-time support helper *per se*.

The purpose of these calls in the VCL source is to ensure that when a form is being streamed to a file, special numeric values are translated into more descriptive textual names. For example, if you look at a new form in text mode (right click on it and choose `View As Text`), its `Color` property will be listed as `clBtnFace`, rather than `$8000000F`. Similarly, if you change the value of the `Cursor` property, it will be listed with the descriptive name, rather than its numeric value.

Streaming is not limited to design-time. Consider when your application runs, and you will realise that the linked in form resource streams have to be read and turned back into real forms. Streaming support must be present at runtime as well as at design-time.

`RegisterIntegerConsts` is a useful routine, but it only affects how property values are streamed. You will also need to write a property editor to get the result you seek.

Having isolated the problem, let's run through how to use `RegisterIntegerConsts` and write a

property editor for a given property of a test component. The component, TTestComponent, is shown in Listing 3 and can be found in the TestComp.pas unit, along with a subrange type TTestRange. The component has a property Test of type TTestRange.

There are two goals with the component. The first is that a Test property value of 1 should be streamed out as trOne (and similarly 2 should be streamed as trTwo). Note that the default value of 0 (trZero) will not be streamed thanks to the default directive used in Listing 3. The other goal is that the Object Inspector should display a Test property value of 0 as trZero (and similarly 1 and 2 should be displayed as trOne and trTwo respectively).

Firstly the streaming goal, which is accomplished with RegisterIntegerConsts. Listing 4 shows the implementation and initialization sections of the TestComp.pas unit.

The initialization section calls RegisterIntegerConsts, passing three parameters. The first is a pointer to the RTTI information for the integer type in question. The other two parameters are references to a pair of translation routines, IdentToTestValue and TestValueToIdent. You can see in the listing that these routines do very little except call VCL helper routines IdentToInt and IntToIdent (introduced in Delphi 3).

➤ *Listing 3: A test component with special property value constants.*

```
type
  TTestRange = 0..2;
const
  trZero = TTestRange(0);
  trOne  = TTestRange(1);
  trTwo  = TTestRange(2);
type
  TTestComponent = class(TComponent)
  private
    FTest: TTestRange;
  published
    property Test: TTestRange read FTest write FTest default trZero;
  end;
```

➤ *Listing 4: Ensuring special numbers are streamed out by name.*

```
const
  TestValues: array[TTestRange] of TIdentMapEntry = (
    (Value: trZero; Name: 'trZero'),
    (Value: trOne;  Name: 'trOne'),
    (Value: trTwo;  Name: 'trTwo'));
function IdentToTestValue(const Ident: String; var TestValue: Integer): Boolean;
begin
  Result := IdentToInt(Ident, TestValue, TestValues);
end;
function TestValueToIdent(TestValue: Integer; var Ident: String): Boolean;
begin
  Result := IntToIdent(TestValue, Ident, TestValues);
end;
initialization
  RegisterIntegerConsts(TypeInfo(TTestRange), IdentToTestValue, TestValueToIdent)
end.
```

➤ *Listing 5: Textual version of the streamed component.*

```
object TestComponent1: TTestComponent
  Test = trOne
  Left = 64
  Top = 40
end
```

➤ *Listing 6: Textual identifier helper routines.*

```
procedure GetTestRangeValues(Proc: TGetStrProc);
var
  I: Integer;
begin
  for I := Low(TestValues) to High(TestValues) do
    Proc(TestValues[I].Name);
end;
```

These two routines from the Classes.pas unit are not documented in the online help but, given an array of TIdentMapEntry records, they translate to and fro between numbers and their string representations. TIdentMapEntry is a record type introduced in Delphi 3, to save defining a new record each time you want to register new integer constants.

Another unit is supplied as the component registration unit: TestReg.pas. This simply registers the component. To test the integer streaming, add TestComp.pas and TestReg.pas into a new package and press the package editor's Install button. This puts the new component onto the Clinic page of the Component Palette.

Drop an instance of the component onto a new form, change the Test property to 1 or 2, then right click on the form and choose View As Text. The component should be described similar to Listing 5. Notice the property value is the readable trOne, rather than simply 1.

Now onto the property editor. Before diving in, the component unit needs an extra utility routine that will be used by the property editor, and can also be used by anything else that needs it. Listing 6 shows the procedure that loops through the private TIdentMapEntry array passing the Name field to the supplied procedure.

To make the component look more finished, the property editor needs to look like Figure 4. It will have a list of available valid values, represented as constant names in an unsorted list. The user can either choose values from the list, or type them in manually. When typing, the user can either enter a constant name, or enter a literal number. For example, if the user enters the number 2, the property editor will display trTwo.

The code to accomplish this is shown in Listing 7. The GetAttributes method requests a property editor with an unsorted list, which works when multiple components are selected. GetValue attempts to translate the property value into a textual name, but will

use a text version of the number if it fails.

The property editor uses `GetValues` to fill the drop-down list with constant names. `GetValues` simply calls `GetTestRangeValues` from Listing 6. When the user enters a new value, `SetValue` is called and passed the string of characters that the user entered. This checks whether the value is an integer constant, using `IdentToTestValue`. If it is, the resultant number is passed to `SetOrdValue`, otherwise the originally entered string is deemed to represent an integer and so is passed to the inherited version of `SetValue`, which will call `StrToInt` and pass the result to `SetOrdValue`.

Scrolling System Tray Text

Q Do you know if it is possible to write text in the Windows System Tray? The objective I'm trying to reach is to get something like the system clock, but displaying text messages that scroll across a small area. I've followed the article from *The Delphi Magazine* (by Marco Cantù in Issue 12) on tray icons, but there does not seem to be an API call that will display text other than the hint.

A Since system tray icons have already been covered in an earlier issue, we need not go back over the same ground. Instead, given an understanding of system tray icons, we need to work out how to get scrolling text to appear.

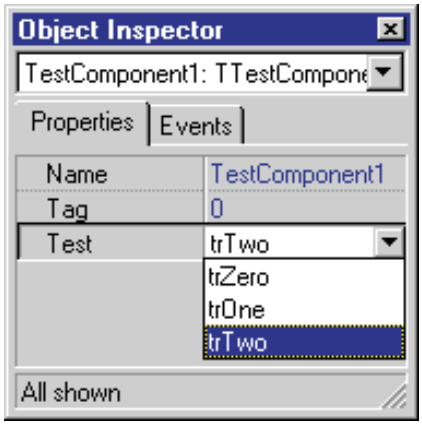
The solution I have come up with involves using an icon as normal, but writing the desired text across it. A timer tick will prompt the icon to be cleared and the text written again with the origin moving leftwards, and the system tray to be updated with the modified icon. This will have the effect of scrolling text. If we take care to reset the origin back to the right side of the icon once the text has 'fallen off' the left side of the icon, the scrolling text will keep wrapping around, giving a never-ending scrolling message.

It should be mentioned that writing text on an icon is not particularly easy, given that it has no canvas. Consequently, I've taken a roundabout route here. I'm writing text on a bitmap, adding the bitmap to an image list and asking the image list to create an icon out of the bitmap.

The code for an application which accomplishes the requirements is shown almost in its entirety in Listing 8, so let's take a look through it.

When the form is created (in `FormCreate`) the `TIcon` and `TBitmap` are created. The bitmap is set the same size as a small icon and has various attributes set, including the background colour. This is set to `clWindow` (which is white by default) and this colour will be made transparent in the resultant icon.

When drawing text on the bitmap, the origin (and therefore the text) moves to the left. To make the text start scrolling in from the right hand side, the starting position (`DrawOffset`) is set to the right



➤ *Figure 4: The new property editor displaying constant names.*

hand side of the bitmap. After setting up the dimensions of the image list (the same as the bitmap), `ImgListToSysTray` is called to add the (currently blank) icon onto the system tray.

`ImgListToSysTray` empties any detritus from the image list, then adds the bitmap to it, turning the background transparent. The same image is then extracted from the image list as an icon, whose handle can be used to set one of the fields of a `TNotifyIconData` record. Other fields are also set to identify the icon (using the form's window handle and an ID number) and specify which additional fields in the record are valid. In this case, we are setting an icon and a tooltip, so the appropriate fields are given sensible values.

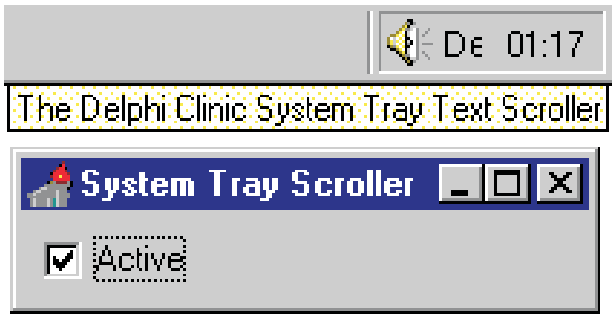
The record is then passed to `Shell_NotifyIcon`. The first time it is called, the bitmap is blank, so we get a transparent icon added (the `NIM_ADD` flag passed to `ImgListToSysTray` adds an icon).

➤ *Listing 7: The property editor class and registration.*

```
type
  TTestProperty = class(TIntegerProperty)
  public
    function GetAttributes: TPropertyAttributes; override;
    function GetValue: string; override;
    procedure GetValues(Proc: TGetStrProc); override;
    procedure SetValue(const Value: string); override;
  end;
function TTestProperty.GetAttributes: TPropertyAttributes;
begin
  //Request list of values
  Result := [paMultiSelect, paValueList, paRevertable];
end;
function TTestProperty.GetValue: string;
begin
  //Try and get nice textual name
  if not TestValueToIdent(TTestRange(GetOrdValue), Result)
    then
    Result := IntToStr(GetOrdValue)
end;
procedure TTestProperty.GetValues(Proc: TGetStrProc);
begin
  //Call subrange type helper routine
  GetTestRangeValues(Proc);
end;
procedure TTestProperty.SetValue(const Value: string);
var
  NewValue: Longint;
begin
  //Try and translate from textual name
  if IdentToTestValue(Value, NewValue) then
    SetOrdValue(NewValue)
  else
    //Otherwise use numeric value
    inherited SetValue(Value);
end;
procedure Register;
begin
  RegisterComponents('Clinic', [TTestComponent]);
  RegisterPropertyEditor(TypeInfo(TTestRange),
TTestComponent, 'Test', TTestProperty)
end;
```

```
type
  TForm1 = class(TForm)
    chkActive: TCheckBox;
    ImageList: TImageList;
    Timer: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure TimerTimer(Sender: TObject);
    procedure chkActiveClick(Sender: TObject);
  private
    Bmp: TBitmap;
    Icon: TIcon;
    TextWidth,
    DrawOffset: Integer;
    procedure ScrollText;
    procedure ImgListToSysTray(Operation: DWord);
  end;
...
const
  ScrollingText =
   'The Delphi Clinic, only in The Delphi Magazine.';
procedure TForm1.ScrollText;
begin
  //Refill bitmap with white
  Bmp.Canvas.FillRect(Rect(0, 0, Bmp.Width, Bmp.Height));
  //Draw text from starting offset
  Bmp.Canvas.TextOut(DrawOffset, 0, ScrollingText);
  //Move offset leftwards
  Dec(DrawOffset, 2);
  if DrawOffset <= -Textwidth then
    //If at end of text, reset offset
    DrawOffSet := Bmp.Width;
  ImgListToSysTray(NIM_MODIFY);
end;
procedure TForm1.ImgListToSysTray(Operation: DWord);
const
  ClinicID = 100;
var
  BmpIndex: Integer;
  NID: TNotifyIconData;
begin
  // Clear image list and add bitmap, with background made
  // transparent
  ImageList.Clear;
  BmpIndex :=
    ImageList.AddMasked(Bmp, Bmp.Canvas.Brush.Color);
  ImageList.GetIcon(BmpIndex, Icon);
  //Setup TNotifyIconData record
  FillChar(NID, SizeOf(NID), 0); //Clear record
  NID.cbSize := SizeOf(NID); //Set byte count field
  NID.Wnd := Handle; //Set owner
  NID.uID := ClinicID; //Set icon ID
  //Identify which other fields are valid
  NID.uFlags := NIF_ICON or NIF_TIP;
  NID.hICon := Icon.Handle; //set icon handle
  //Set tooltip
  NID.szTip :=
    'The Delphi Clinic System Tray Text Scroller';
  //Set icon in system tray
  Win32Check(Shell_NotifyIcon(Operation, @NID));
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  //Create icon
  Icon := TIcon.Create;
  //Set up bitmap
  Bmp := TBitmap.Create;
  Bmp.Width := GetSystemMetrics(SM_CXSMICON);
  Bmp.Height := GetSystemMetrics(SM_CYSMICON);
  Bmp.Canvas.Brush.Color := clWindow;
  Bmp.Canvas.Font.Name := 'Verdana';
  Bmp.Canvas.Font.Size := 10;
  Bmp.Canvas.Font.Color := clWindowText;
  TextWidth := Bmp.Canvas.TextWidth(ScrollingText);
  //Start text off at right-hand side of bitmap
  DrawOffSet := Bmp.Width;
  //Set up image list
  ImageList.Width := Bmp.Width;
  ImageList.Height := Bmp.Height;
  ImgListToSysTray(NIM_ADD);
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  //Remove icon from system tray and tidy up
  ImgListToSysTray(NIM_DELETE);
  Bmp.Free;
  Icon.Free;
end;
procedure TForm1.TimerTimer(Sender: TObject);
begin
  ScrollText
end;
procedure TForm1.chkActiveClick(Sender: TObject);
begin
  Timer.Enabled := chkActive.Checked
end;
```

➤ *Listing 8: Scrolling text on a system tray icon.*



➤ *Figure 5: The system tray icon, scolling its text.*

A timer ticks every 50 milliseconds causing `ScrollText` to be called. This simple method fills the bitmap with white background and draws the text at the current offset, before shifting the offset along (it wraps back to the start if it goes too far). Another call to `ImgListToSysTray` with a `NIM_MODIFY` parameter replaces the old icon with the new one. With this happening 20 times per second, the scrolling looks acceptable.

A checkbox on the form allows the scrolling to be stopped and started as the user likes. The event handler simply toggles the state of the timer's `Enabled` property. Finally, when the form is destroyed, the bitmap and icon objects are freed and the icon is removed from the system tray.

You can see (well, almost) the program running in Figure 5. The tooltip and form are clearly visible and, with some imagination, you can see where the message text scrolls across (the word *Delphi* is just starting to be displayed). To save your imagination, just load the SysIcon.dpr project on this month's disk and run it in any 32-bit version of Delphi.

Acknowledgements

Thanks go to Steve Axtell from Inprise for the InterBase and BDE help this month.