The Delphi CLINIC

Edited by Brian Long

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

String Grid Query

Q I would like to use a string grid to display some data, but I do not want the user to be able to select any cells. Setting the `Enabled` property to `False` sort of works, but the initial selected cell is still selected and the grid's scroll bars stop working. Any ideas?

A To stop the user selecting a different cell, you can make an `OnSelectCell` event handler for the grid and insert the statement:

```
CanSelect := False
```

However, despite not being able to select a new individual cell, the grid still supports highlighting cell ranges with the mouse. To disable this, use the grid's `Options` set property and exclude `goRangeSelect`. This can be done in the Object Inspector by expanding `Options` and setting `goRangeSelect` to `False` or in code as:

```
with StringGrid1 do
  Options :=
    Options - [goRangeSelect]
```

Additionally, to stop the initial highlighting of the first cell in the grid, you need to assign a value to the grid's `Selection` property (a `TGridRect`). Co-ordinates of -1 do the trick. The sample project GRID.DPR on the disk does exactly this with a 10 by 10 grid to prove the point (see Listing 1).

ComboBox Width Anomaly

Q If a Delphi 1 `TComboBox`'s `Width` property is adjusted wider at run-time its dropped down rectangle is also made wider. However, as the `Width` property is reduced the dropped down rectangle remains at the widest setting. I cannot find a way of managing the width of the drop-down dynamically. Is this a Windows "feature" and if so is there an appropriate API call to control it?

A In Win32 there is a `cb_SetDroppedWidth` message you can send to the combobox, used like this:

```
ComboBox1.Perform(
  cb_SetDroppedWidth,
  WidthValue, 0)
```

but then again in Delphi 2 and 3 the problem does not occur. I can find no support for this in Win16. However, you can achieve the desired effect in Delphi 1 by calling

```
THintWindow(
  ComboBox1).ReleaseHandle
```

just after changing `Width`. This is the old cheeky typecast trick as I've previously expounded several times in this column *[check Brian's articles in Issues 3, 4 and 5 for additional pearls of typecasting wisdom. Editor]*.

Poor Person's Polymorphism

Q I have a program in which I wish to call one of several procedures dependent upon a variable. Rather than use a big case statement to evaluate the variable (which means using an ordinal type for the variable) I would like to be able to pass a string constant to a handler procedure and for the handler to call the procedure of the name defined by that constant.

A This desire of yours is not very far removed from polymorphism. When you call a polymorphic routine you call a specific procedure and, depending on some circumstance (ie which object it is), code executes to call one of a variety of subroutines using some form of lookup system. You want to call a routine and, depending on some circumstance (the value of a string), cause one of a variety of routines to be called. Much the same thing but without the objects. Poor person's polymorphism.

This is typically implemented using an array of procedural variables. You will not be able to map

➤ *Listing 1*

```
procedure TForm1.FormCreate(Sender: TObject);
{ Initialise selection record with -1s }
const
  NoSelection: TGridRect = (Left: -1; Top: -1; Right: -1; Bottom: -1);
var
  LoopX, LoopY: Integer;
begin
  with StringGrid1 do begin
    { Fill grid with dummy values }
    for LoopX := 1 to ColCount do
      for LoopY := 1 to RowCount do
        Cells[LoopX, LoopY] := Format('(%d,%d)', [LoopX, LoopY]);
    { Get rid of default selection }
    Selection := NoSelection;
  end
end;
procedure TForm1.StringGrid1SelectCell(Sender: TObject;
  Col, Row: Longint; var CanSelect: Boolean);
begin
  CanSelect := False
end;
```

directly from the string constant to a procedure name because at runtime the names of the procedures have typically gone. Delphi is a compiler not an interpreter. So instead you could use values from an enumerated type which would allow you to index into the procedural variable array. The POORPER.DPR project on the disk shows an example of doing exactly this (Listing 2).

There is a simple form with three buttons and each button has an event handler that invokes a different routine by calling the handler procedure with a different value. Note the use of a typed constant to set up the array of procedural variables. Each element of the array has a different one of the target routines assigned to it. The intended subroutine gets called by a simple array access. Because the array contains procedural variables, this invokes the routine.

The array is declared to be of type `TProcedure`, a pre-defined type for parameterless procedures. If you need more exotic procedures and functions, you should define another procedural type. For example, for a procedure taking a constant string parameter:

```
TParamProc =
  procedure(const S: String);
```

The call in the handler routine would then need a parameter list afterwards. Maybe something like Listing 3 (POORPER2.DPR).

Early Query Termination

Q How can one interrupt an SQL query against an Oracle database which has gone astray? For example the query has been running for 15 minutes and you want to let the user cancel out.

A There is nothing in the BDE to help directly: a three fingered salute is probably as good as anything here. With InterBase you could shut down the database, but this would affect everybody connected to it. You coluld ask Oracle if there is any way to safely terminate that user's connection.

It is perhaps best to prevent this kind of problem from occurring in the first place. If you are using the 32-bit BDE then you can set up `MAX ROWS` in the BDE Configuration application. For example, if this was set to 1000, the BDE would only fetch back the first 1000 records of any query. The minimum value for this setting must be sufficient for any schema information to be queried back to the client.

Platform Checking

Q How do I determine which platform my program is running on?

A You don't specify which product version you are compiling with. If you are using Delphi 2 or 3, you can use the `SysUtils` variable `Win32Platform`. This will have one of two values: either `Ver_Platform_Win32_Windows` or `Ver_Platform_Win32_NT`. If you insist on using the Windows API you have a choice of `GetVersion` or `GetVersionEx`.

If you are using Delphi 1, you are forced to use `GetVersion`. The problem with this is that the documentation does not show how to distinguish between Windows 3.1x or Windows NT: both give a version of 3.10. Fortunately the *Microsoft Developer Network Library* CD (the API programmer's first port of call) helps out. There is an otherwise undocumented flag that can be located which is set under NT. Listing 4 shows an appropriate function along with a sample call to it (from TESTVER.DPR on the disk).

Going back to the 32-bit versions of Delphi, Delphi 3 adds several new variables to accompany `Win32Platform`. We can now also get the major and minor Windows

➤ *Listing 2*

```
type
  { Type to signify different routines }
  TProcName = (pnProc1, pnProc2, pnProc3);
{ Here are the routines }
procedure One; far;
begin
  ShowMessage('One');
end;
procedure Two; far;
begin
  ShowMessage('Two');
end;
procedure Three; far;
begin
  ShowMessage('Three');
end;
{ Here is the handler routine that calls stuff }
procedure DoIt(ProcToCall: TProcName);
const
  { An array of the routines }
  Procs: array[TProcName] of TProcedure = (One, Two, Three);
begin
  { Call the appropriate routine }
  Procs[ProcToCall];
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  DoIt(pnProc1);
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
  DoIt(pnProc2);
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
  DoIt(pnProc3);
end;
```

➤ *Listing 3*

```
procedure DoIt(ProcToCall: TProcName; const S: String);
const
  { An array of the routines }
  Procs: array[TProcName] of TParamProc = (One, Two, Three);
begin
  { Call the appropriate routine }
  Procs[ProcToCall](S);
end;
```

versions along with the build number and a possible additional information string. See Listing 5.

Bad Double-Clicks

Q My users often double-click on speed buttons when in truth they should only single click. Because one of the speed buttons loads another form which also has a speed button in almost the same place, the second click gets passed to that component – with disastrous effects! Can I get the second click and ditch it, pull it out of the message queue altogether?

A There may be a better way, but one possibility is to delay the second form for 2 tenths of a second (or maybe more) to avoid the problem. See Listing 6.

Setting The Time And Date

Q Delphi has lots of support for formatting dates and times, but I can't find anything that lets me set the date and time.

A The Win32 API is `SetLocalTime`. The functions in Listing 7 work quite nicely on my Windows 95 machine. There is a possibility that NT will require you to enable your privilege to set the date and time, but this should give you a start. Listing 8 has Delphi 1 equivalents, using interrupt calls in assembler (this is how `SysUtils` gets the information in the first place), from the `TimeSet` unit on the disk.

➤ Listing 8

```
procedure SetTime(Time: TDateTime);
var Hour, Min, Sec, HSec: Word;
begin
  DecodeTime(
    Time, Hour, Min, Sec, HSec);
  HSec := HSec div 10;
  asm
    mov ch, Byte(Hour)
    mov cl, Byte(Min)
    mov dh, Byte(Sec)
    mov dl, Byte(HSec)
    mov ah, $2D
    int $21
  end;
end;
procedure SetDate(Date: TDateTime);
var Year, Month, Day: Word;
begin
  DecodeDate(Date,Year,Month,Day);
  asm
    mov cx, Year
    mov dh, Byte(Month)
    mov dl, Byte(Day)
    mov ah, $2B
    int $21
  end;
end;
procedure SetNow(
  DateTime: TDateTime);
begin
  SetDate(Int(DateTime));
  SetTime(Frac(DateTime))
end;
```

Chasing Carets

Q In Issue 9, you described a 16-bit Windows problem when tabbing between edit controls. If one edit has an `OnExit`

➤ Listing 4

```
function WindowsPlatform: TVersion;
const wf_WinNT = $4000;
begin
  if GetWinFlags and wf_WinNT <> 0 then
    Result := WinNT
  else if HiByte(LoWord(GetVersion)) = 95 then
    Result := Win95
  else
    Result := Win16 { or unidentifiable version }
    { actual version number can be obtained with
      low and high words of GetVersion result }
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  case WindowsPlatform of
    Win16: Caption := 'Windows 3.x';
    WinNT: Caption := 'Windows NT';
    Win95: Caption := 'Windows 95';
  end;
end;
```

➤ Listing 5

```
procedure TForm1.FormCreate(Sender: TObject);     { project D3WINVER.DPR on disk }
begin
  case Win32Platform of
    Ver_Platform_Win32s: //Unlikely, but still...
      lblPlatform.Caption := 'Win32s';
    Ver_Platform_Win32_Windows:
      lblPlatform.Caption := 'Windows';
    Ver_Platform_Win32_NT:
      lblPlatform.Caption := 'Windows NT';
  end;
  lblVersion.Caption := Format('%d.%d', [Win32MajorVersion, Win32MinorVersion]);
  lblBuild.Caption := Format('%x', [Win32BuildNumber]);
  lblAdditionalInfo.Caption := Win32CSDVersion;
end;
```

➤ Listing 6

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
var OldTime: TDateTime;
begin
  OldTime := Time;
  { Delay for 0.2 seconds }
  repeat until Time >= OldTime + 0.2 / SecsPerDay;
  { Swallow any second click }
  Application.ProcessMessages;
  { Now show the other form, after the other click has been eaten }
  Form2.Show;
end;
```

➤ Listing 7

```
procedure SetNow(DateTime: TDateTime);
var ST: TSystemTime;
begin
  with ST do begin
    DecodeDate(DateTime, wYear, wMonth, wDay);
    DecodeTime(DateTime, wHour, wMinute, wSecond, wMilliSeconds);
  end;
  if not SetLocalTime(ST) then
    raise Exception.Create(SysErrorMessage(GetLastError));
end;
procedure SetTime(Time: TDateTime);
begin
  //Use passed in time + system date
  SetNow(Time + Date)
end;
procedure SetDate(Date: TDateTime);
begin
  //Use passed in date + system time
  SetNow(Time + Date)
end;
```

handler that brings up a message box or other modal form, the next one refuses to display any input caret or highlight any text when the modal window is closed, despite being the focused control. Using Alt-Tab to switch away from the app and then back to it fixes the problem. You suggested ensuring focus is left on the original edit control to alleviate the problem. This is probably fine if you are alerting the user to some error in the value in the first edit, but what about if the message is purely informative? Is there a way to allow an edit to bring up a message box on exit and not get the lost caret problem?

A It should be noted that the problem occurs when you change focus to an edit from any control, where that control has an OnExit event as described. The original suggestion was to support error messages in on-tab validation for edit controls. The OnExit event handler can look something like this:

```
procedure TForm1.Edit1Exit(
  Sender: TObject);
begin
  if BogusValue then
  begin
    ShowMessage('Bad value!');
    (Sender as TEdit).SetFocus
  end
end;
```

Having pondered the problem for a while, if you do want focus to go wherever the user was planning, it seems you have to delve rather deeper. The problem of the lost caret comes about because the OnExit event seems to be triggered whilst the original edit is processing its wm_KillFocus message and the next control is processing its wm_SetFocus message. The OnExit handler rips focus from the new control by virtue of a modal window. This causes a wm_KillFocus to be sent to the next control before it has finished the wm_SetFocus processing and is sufficient to stun the 16-bit edit code into visual silence.

To remedy the problem (and I warn you this isn't very elegant) you can take over the offending control's underlying window procedure and manually call the OnExit handler *after* the message processing has completed. To accomplish this, set up your controls with their OnExit handlers set up as normal.

Now, to stop the OnExit handlers being triggered incorrectly, use the Events page of the Object Inspector to disassociate the handler from the event. In other words, select the OnExit event in the Object Inspector, highlight the value (the procedure name) and delete it. This is important: if you don't do this, the OnExit handler will trigger twice. Once due to the normal VCL mechanics and once due to this extra code.

Now you need a data field to hold the window procedure of the control, and you will need to declare and implement a form method that will act as the replacement window procedure. Lastly, the OnCreate event handler of the form needs to set up the new window procedure and the OnDestroy needs to put things back to how they were.

The new window procedure only does things different from the norm if a wm_KillFocus message is received (in other words the control is losing focus). After doing the normal processing, it calls the OnExit event handler explicitly. Listing 9 has the code where Edit1 has an OnExit handler (although it doesn't show up in the Object Inspector: see above) and Edit2 previously displayed the lost caret (or didn't display it, if you see what I mean).

If you were happy to write new components then this window procedure replacement could be replaced by a simple message handler in the new component.

Acknowledgements

Thanks to Steve Axtell for his help this month.

➤ *Listing 9*

```
unit Caretu;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls;
type
  TForm1 = class(TForm)
    Edit2: TEdit;
    Edit1: TEdit;
    { Set up the OnExit handler as normal, then use the
      Object Inspector to disassociate the handler from the
      event, as it will be called indirectly }
    procedure Edit1Exit(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Have a data field to store the old window procedure }
    FOldWndProc: TFarProc;
    { Declare a new surrogate window procedure }
    procedure NewWndProc(var Message: TMessage);
  end;
var Form1: TForm1;

implementation
{$R *.DFM}

{ Here's the routine that was the normal event handler }
{ Of course it's now an abnormal event handler }
procedure TForm1.Edit1Exit(Sender: TObject);
begin
  ShowMessage('Hello world!');
end;

{ Here's the surrogate window procedure }
procedure TForm1.NewWndProc(var Message: TMessage);
begin
  with Message do begin
    { Do message processing }
    Result := CallWindowProc(FOldWndProc, Handle, Msg,
      WParam, LParam);
    { _After_ processing call the event handler manually }
    if Msg = wm_KillFocus then
      Edit1Exit(Edit1);
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  { Set up replacement window procedure }
  FOldWndProc := TFarProc(SetWindowLong(Edit1.Handle,
    gwl_WndProc, Longint(MakeObjectInstance(NewWndProc))));
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  { Tidy up replacement window procedure }
  FreeObjectInstance(Pointer(SetWindowLong(Edit1.Handle,
    gwl_WndProc, LongInt(FOldWndProc))));
end;

end.
```