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

What Tasks Are Running?

Q How can I find out which programs are running on my users' machines?

A To find currently loaded programs requires using some `ToolHelp` routines. Which ones you use depends on what you want to find. It seems there are four common requirements: finding the names of all running tasks (ie programs), finding the executable file names of all running tasks, finding the names of all modules (ie tasks and DLLs, including fonts and .DRV drivers) and finding the executable file names of all modules. Listing 1 includes four routines which do all these, taken from the unit PROGRAMS.PAS on the disk. Just pass in a `TStrings` object as a parameter to any of them, and it will be filled with the relevant names. So to fill up a listbox, for example, you could call `GetModuleExes(Listbox1.Items)`.

Intercepting Keystrokes

Q How do I trap the `Enter` key in edit boxes and do something like treat it as a `Tab` keypress and move to the next control?

A Firstly, you need to decide whether this should apply all over a form or in a particular control or set of controls (the control type is not important, although see below). If for the form, then set the form's `KeyPreview` property to `True` and do the work in the form's `OnKeyPress` handler. If for one control, do it in the control's `OnKeyPress` handler, and if for a set of controls, make them all share an `OnKeyPress` handler (if the target action is the same). It's worth mentioning at this point that a focused button, or any button with its `Default` property set to `True` will always take the `Enter` key to be the equivalent of clicking it. If you want to use the `Enter` key to navigate around your form, set all the `Default` properties to `False`.

Normally, we want `Enter` to go round the natural tab order, but we may also want `Shift-Tab` to go round in reverse. We can detect the state of the `Shift` key at any time using the `GetKeyState` API. If the high bit is set, it is currently pressed. The code in Listing 2 will do the trick. Note that we test the `Key` parameter against ASCII versions of the virtual key code we are looking for. Also, `Key` gets set to a zero character (there is no key with a virtual key code of zero) if we want to hide or swallow the key, ie we do not wish normal processing (often a beep) to occur.

Opening And Closing Maximized MDI Children In Windows 3.1x

Q Running under Windows 3.1, the MDI project from the Project Template Gallery exhibits a problem. If you choose `File | New`, maximise this child window and double-click its system box (to the

➤ *Listing 1*

```
procedure GetTaskModules(Strings: TStrings);
var TE: TTaskEntry;
begin
  Strings.Clear;
  TE.dwSize := SizeOf(TE);
  if TaskFirst(@TE) then repeat
    Strings.Add(StrPas(TE.szModule));
  until not TaskNext(@TE);
end;
procedure GetTaskExes(Strings: TStrings);
var TE: TTaskEntry;
    ME: TModuleEntry;
begin
  Strings.Clear;
  TE.dwSize := SizeOf(TE);
  ME.dwSize := SizeOf(ME);
  if TaskFirst(@TE) then repeat
    ModuleFindHandle(@ME, TE.hModule);
    Strings.Add(StrPas(ME.szExePath));
  until not TaskNext(@TE);
end;
procedure GetModules(Strings: TStrings);
var ME: TModuleEntry;
begin
  Strings.Clear;
  ME.dwSize := SizeOf(ME);
  if ModuleFirst(@ME) then repeat
    Strings.Add(StrPas(ME.szModule));
  until not ModuleNext(@ME);
end;
procedure GetModuleExes(Strings: TStrings);
var ME: TModuleEntry;
begin
  Strings.Clear;
  ME.dwSize := SizeOf(ME);
  if ModuleFirst(@ME) then repeat
    Strings.Add(StrPas(ME.szExePath));
  until not ModuleNext(@ME);
end;
```

left of the File menu) it doesn't close, whereas when restored, it does. This problem does not occur in Windows 95. Also, if I set the MDI child's WindowState property to wsMaximized at design time, choosing File | New causes a normal window to be created which is then quite clearly maximised. This makes the program look rough around the edges (even in Windows 95). Can this be fixed?

A Let's take the latter problem first, since it's easier. The reason the MDI child is created restored and then maximised is due to certain unspecified problems that can occur if you make it maximised straight off. The snippet from TForm.CMShowingChanged in FORMS.PAS shown in Listing 3 gives the clue.

To avoid the flickering you get on screen, Neil Rubenking (email CompuServe 72241,50) suggests changing the body of the project's TMainForm.CreateMDIChild (in the file MAIN.PAS) from:

```
{ create a new MDI child window }
Child :=
 TMDIChild.Create(Application);
Child.Caption := Name;
```

to:

```
LockWindowUpdate(Handle);
{ create a new MDI child window }
Child :=
 TMDIChild.Create(Application);
Child.Caption := Name;
LockWindowUpdate(0);
```

As for the other problem, this has been subject of debate since Windows 3.0, where there was a bug. The bug was fixed, but not completely, in Windows 3.1. Principally, the problem won't occur if the MDI client (TForm.ClientHandle) is the first child in the MDI window, but it isn't in a Delphi application with a speedbar and/or status line. Two solutions were presented in an online discussion of this problem dating back to May 1995. With the authors' permissions I am reproducing them here.

➤ *Listing 2*

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if Key = Chr(vk_Return) then begin
    Key := #0;
    SelectNext(ActiveControl, GetKeyState(vk_Shift) and $80 = 0, True);
  end;
end;
```

➤ *Listing 3*

```
if FormStyle = fsMDIChild then begin
  { Fake a size message to get MDI to behave }
  if FWindowState = wsMaximized then begin
    SendMessage(Application.MainForm.ClientHandle,
                WM_MDIRESTORE, Handle, 0);
    ShowWindow(Handle, SW_SHOWMAXIMIZED);
  end
```

➤ *Listing 4*

```
procedure TMainForm.WMNCLButtonDblClk(var Msg: TWMNCLButtonDblClk);
var tScreenPt, ClientPt: TPoint;
begin
  ScreenPt.X := Msg.XCursor;
  ScreenPt.Y := Msg.YCursor;
  ClientPt := ScreenToClient(ScreenPt);
  if (Msg.HitTest = htMenu) and (ClientPt.X < GetSystemMetrics(sm_CYMenu))
    and (ActiveMDIChild <> nil) and
    (ActiveMDIChild.WindowState = wsMaximized) then begin
    ActiveMDIChild.Close;
    Msg.Result := 0;
  end else
    inherited;
end;
```

The first solution involves modifying the VCL. Find this line in TForm.AlignControls in the file FORMS.PAS:

```
SetWindowPos(FClientHandle,
  HWND_BOTTOM, Left, Top,
  Right - Left, Bottom - Top,
  SWP_NOCOPYBITS);
```

Now change the HWND_BOTTOM to HWND_TOP. The second solution requires no VCL changes. In the private section of your main MDI form class add:

```
procedure WMNCLButtonDblClk(
 var Msg: TWMNCLButtonDblClk);
 message WM_NCLButtonDblClk;
```

Now add the code in Listing 4 to the implementation section.

Converting Times To Floats

Q If I have a TTimeField in a table, I can get the time as a floating point number by using the AsFloat method. This allows me to add and subtract start and end times. How do I get a floating point version of a normal TDateTime time value? There is no TimeToFloat routine.

A Fortunately there is no need for such a routine. The TDateTime data type is implemented as a Double anyway. The whole part represents the number of days since some starting point and the fractional part is the time as a fraction of a day. In Delphi 1 the number of days was measured from 1st Jan, 0001. Delphi 2 changes this (for OLE Automation compatibility) to be 30th December, 1899. To convert a Delphi 1 TDateTime value to a Delphi 2 value, subtract DateDelta (693594) from the Delphi 1.0 date. This value is the number of days between the Delphi 1 and Delphi 2 starting dates.

For example, consider the date 1st Jan, 1900. In Delphi 1 this gives a TDateTime value of 693596. In Delphi 2 this gives a value of only 2. The Delphi 1 value needs DateDelta taken away to yield the correct TDateTime value in Delphi 2.

Bitmaps For Brushes

Q Why can't I set up a custom bitmap brush? A statement like this fails:

```
Form1.Canvas.Brush.Bitmap.LoadFromFile(
  'Brush.Bmp');
```

A A brush's bitmap starts off as `nil`. If you want to use a custom bitmap, you need to assign a valid `TBitmap` object to the property first. Note that you are also responsible for freeing the bitmap. Listing 5 is a code snippet from the project BRUSH.DPR. There's a section in \DELPHI\MANUALS.TXT which describes how to update a brush once you change a bitmap that a brush is referencing.

Multiple Arguments

Q I am trying to implement a routine that takes an array of values and adds them together. Can you give some help on accepting valid values, including strings and characters that represent numbers.

A This facility of Delphi's, to allow a user-defined routine to take a variable number of arguments (as elements of an array) and let you examine their type and value, is so useful that I am happy to supply another example. The concept of passing an array, built on the fly, to a routine should be familiar to anyone who has called such routines as `TTable.FindKey`, `Format` and `MessageDlg`, but implementing them is not as easy as calling them. When declaring a routine that takes an array of any type of argument, the array is declared to be `const Argument: array of const`. When examining the values of the array (whose bounds go from `Low(Argument)` to `High(Argument)` and must number at least one) their type will be `TVarRec`, a variant record defined, but not explained in depth, in the on-line help and reproduced in Listing 6.

The idea is to examine the `VType` field (whose value will be one of `vtInteger`, `vtBoolean`, `vtChar` etc) and use it to determine what other field to use to get the value of the parameter. So if `VType` is `vtExtended` the field you'd look at is `VExtended`. This is of type `PExtended` – the address of (or pointer to) a floating point number – so the value of the parameter would be `VExtended^`.

The example routine (Listing 7) will accept numbers that are floating points or integers and those represented by strings and

➤ *Listing 5*

```
BrushBmp: TBitmap;
...
BrushBmp := TBitmap.Create;
BrushBmp.LoadFromFile('brush.bmp');
{ A brush's bitmap starts nil - assign it a valid TBitmap }
Brush.Bitmap := BrushBmp;
...
{ A brush doesn't delete its bitmap object }
Canvas.Brush.Bitmap := nil;
BrushBmp.Free;
```

➤ *Listing 7*

```
function Add(const Values: array of const): Double;
var    Loop: Byte;
const  BoolStrs: array[Boolean] of String[5] = ('False', 'True');
procedure Error(const S: String);
begin
  raise EInvalidOp.Create('Bogus value ' + S);
end;
begin
  Result := 0;
  for Loop := Low(Values) to High(Values) do
    with Values[Loop] do
      case VType of
        vtInteger : Result := Result + VInteger;
        vtBoolean : Error(BoolStrs[VBoolean]);
        vtChar :
          if VChar in ['0'..'9'] then
            Result := Result + Ord(VChar) - Ord('0')
          else
            Error('"' + VChar + '"');
        vtExtended : Result := Result + VExtended^;
        vtString :
          try
            Result := Result + StrToFloat(VString^)
          except
            Error('"' + VString^ + '"')
          end;
        vtPointer : Error(Format('%p', [VPointer]));
        vtPChar : Error(StrPas(VPChar));
        vtObject :
          { Forms don't have a name property value read in }
          if (VObject is TComponent) and not (VObject is TForm) then
            Error(TComponent(VObject).Name + ': ' + VObject.ClassName)
          else
            Error(VObject.ClassName);
        vtClass: Error(VClass.ClassName);
      end;
end;
```

➤ *Listing 6*

```
const
  vtInteger  = 0;
  vtBoolean  = 1;
  vtChar     = 2;
  vtExtended = 3;
  vtString   = 4;
  vtPointer  = 5;
  vtPChar    = 6;
  vtObject   = 7;
  vtClass    = 8;
type
  TVarRec = record
    case Integer of
      vtInteger :
        (VInteger: Longint;
         VType: Byte);
      vtBoolean :
        (VBoolean: Boolean);
      vtChar :
        (VChar: Char);
      vtExtended :
        (VExtended: PExtended);
      vtString :
        (VString: PString);
      vtPointer :
        (VPointer: Pointer);
      vtPChar :
        (VPChar: PChar);
      vtObject :
        (VObject: TObject);
      vtClass :
        (VClass: TClass);
  end;
```

characters. All error values will cause an exception to be raised, displaying the value in question. This allows the example to show how to read all data types. This code is shown being called correctly and incorrectly (in a number of cases) in project VARARGS.DPR.

Table Locks

Q I have an application where several Paradox tables need to be opened with table write-locks prior to multi-table batch updates. Other users need access to the tables in read-only mode (interactive Paradox users, etc) so I can't open the tables for exclusive use for my batch operation. Delphi does not seem to support the concept of table-locks but the BDE does. How can I make use of this functionality?

A The Delphi manual errata file \DELPHI\MANUALS.TXT documents a new type `TLockType` (an enumerated type with values `ltReadLock` and `ltWriteLock`) and also two `TTable` methods that take a `TLockType` argument (`LockTable` and `UnlockTable`) that can be used to apply and release table locks.

This often-missed file also provides a useful routine for printing bitmaps using the Windows API `StrechDIBits` rather than Delphi's `StretchDraw` method. A variant of this is used by the VCL when a form's `Print` method is called: `TForm.Print` calls it with the bitmap returned by `TForm.GetFormImage`.

The Form With No Name

Q When I read a form's `Name` property at run-time, it gives me an empty string. At design time it has a name – in fact it prevents me from giving a blank name. Why?

A At design time, the name is used mainly to create the form variable name and also the form class name. At run-time, when a form is created and reads itself in from the executable it explicitly ignores the name that was stored. Here is a snippet from somewhere deep inside the Classes unit:

```
ReadStr; { Ignore class name }
ReadStr; { Ignore object name }
```

Each component needs a unique name so things like `FindComponent` can be guaranteed to work. Now consider an MDI application where there are five MDI children, all based on the same form type. If any given form had its `Name` property set to what it 'should' be, there would be five components called `MDIChildForm`, or whatever. It's the same with SDI apps. You can instantiate a form as many times as you want. To avoid problems, `Name` is ignored. However, this is only the case in Delphi 1. In Delphi 2, the snippet above changes to:

```
ReadStr; { Ignore class name }
if csDesigning in
  Result.ComponentState then
  ReadStr
else
  Result.Name :=
    FindUniqueName(ReadStr);
```

So `Name` has a valid value at run-time in Delphi 2. If you want the name set up in Delphi 1 (and you know there will only be one of each), in the form's `OnCreate` handler, use, eg:

```
{$ifdef Windows}
  Name := Copy(ClassName, 2,
    Pred(Length(ClassName)));
{$endif}
```