Tips & Tricks

Showing Modal Forms

In August's issue, Richard Smith submitted a routine to help show modal forms by way of a function. This is a great idea, but if simplicity is what you are after, we can shorten the suggested routine to only take one parameter (the second parameter serves little purpose anyway) as shown in Listing 1. Incidentally, making the form class reference a `const` parameter serves little purpose since a class reference is implemented as a pointer anyway.

Contributed by Brian Long.

TDBNavigator Button Click

I read Robert Palomo's article *Chasing Objectivity with TDBNavigator* in the June issue with great interest. I had also been thinking of deriving a descendant of `TDBNavigator`, in my case because I wanted the `Insert` button to append a record rather than inserting one.

The author describes a possible approach: subclass `TDBNavigator`, copy and paste the source of `Click` and `BtnCLick` to the new class and tailor `BtnClick` to our needs. He then notes that `BtnClick` references a private variable `FOnNavClick` (the address of the `OnClick` event, if any, of our navigator component) which we cannot access, and then abandons this approach in favour of another one.

However, further inspection of the VCL source tells us that there is a published property `OnClick` whose very purpose it is to provide access to the private variable `FOnNavClick`. So we can declare a local variable `FOnNavClick` of type `ENavClick` and use the property to set its value within the method:

```
procedure TmbDBNavigator.BtnClick(
  Index: TNavigateBtn);
var FOnNavClick: ENavClick;
begin
  ...
  FOnNavOnClick := OnClick;
  if not csDesigning in ComponentState) and
    Assigned(FOnNavClick) then
    FOnNavClick(Self, Index);
end;
```

If you try this you will see that it will compile but still doesn't work! In fact, the smart linker hasn't even included our `BtnClick` in the executable. The reason is that the only place from which `BtnClick` is called is from our `Click`, and our `Click` is not called from anywhere! The various references to `Click` in `TDBNavigator` still point to the original `Click`, not to ours. So what we need to do is replace these references with references to our own `Click`.

This is not as difficult as it might seem. When we look again at the VCL source we can see that the `Click` method is set as the `OnClick` event routine of the various navigator buttons. This is done by a method called `InitButtons` which is called from the `Constructor`. So what we can do is override the `Constructor` and replace the `OnClick` events for the various navigator buttons with a reference to our own `Click` routine, as shown in Listing 2.

And now it works! Our `BtnClick` method now processes the navigator buttons and we can make it do all sorts of clever things Borland never dreamt of! Admittedly this approach is still not 100% ideal (as Robert Palomo notes, we should have been able to override the original `BtnClick` in the first place!) but is is a lot better than duplicating all `TDBNavigator`'s VCL source and creating our own navigator component from scratch. The code for the new navigator is on this month's disk as file MBNAV.PAS.

Contributed by Maarten van den Broek from The Netherlands

Laptop Battery Status

In most of my software I include the unit shown in Listing 3.

The `PowerMan` function returns -1 if the laptop does not have APM, otherwise it returns a value in the range 0 to 100 which indicates the percentage power remaining in the battery. This is nice because you can make a bar graph or a header showing battery condition automatically when `PowerMan` detects a value 0 to 100. If no APM is present, you can just set the `Visible` property of the bargraph to `False`. It really impresses customers to have the software tell them about their laptop's battery condition!

Contributed by mike pijl, mike_pijl@mindlink.bc.ca

TStringList Versus TStringCollection

Someone contacted me recently who had been working extensively with `TStringCollection` from Borland Pascal 7.0 to read strings from text files. They found that trying to do the same with Delphi's `TStringList.ReadFromFile` took about 5 times as long and even using `TList` didn't improve the result. With a 350Kb file, the constant hard disk grinding indicated

➤ *Listing 1*

```
procedure CreateModal(FormClass: TFormClass);
begin
  with FormClass.Create(Application) do
    try
      ShowModal
    finally
      Free
    end
end;
```

that a lot of swap file space was also being used, which was not the case with TStringCollection. This happens with both Delphi 1 and 2.

Well, it turns out that the internal implementation of TStringList is a bit different between Delphi 1 and 2, but they both suffer from the same problem: they grow by a constant, hard coded, number of items when the array of pointers/records needs to be expanded. In the old (and in my opinion better implemented) TCollection objects you had to specify a delta when calling the constructor. In the new Delphi TList and TStringList classes this delta is kindly decided for you and you cannot override it! The delta value in Delphi 1 and 2 is 16 for both TList and TStringList. So, whenever the current capacity of the pointer/record array is less than needed, the array is re-allocated to expand it by 16 items. The re-allocation usually means allocating a new memory block, copying the old block into the new one and de-allocating the old block (Delphi 2 can re-allocate inline if conditions allow). If you try to fill a list with 16,000 items, this re-allocation is therefore done 1,000 times! On average, a 32Kb block is copied each time (16000 / 2 * SizeOf(Pointer)). This means that 32Mb of memory is being copied. In Delphi 2 this effect is even worse for TStringLists because they are implemented as arrays of records rather than arrays of pointers. The record size is 8 bytes, so the corresponding number would be 64Mb!

TStringList has Grow and SetCapacity methods, but these are private and thus cannot be accessed. In Delphi 1 the same applies to TList, while in Delphi 2 TList has a protected, virtual Grow method (that can be overriden by descendants) and a public Capacity property to set the capacity directly (and so avoid the constant allocation, move, de-allocation cycles).

I would not recommend using TStringList for high volume, time-sensitive work. If Borland had implemented it a tiny bit more flexibly and consistently (TList is fixed in Delphi 2, but not TStringList!) this would not have been a problem. If you have the VCL source you can use it as a starting point for your own version of TStringList that fixes this problem.

Contributed by Hallvard Vassbotn, hallvard@falcon.no

➤ *Listing 2*

```
constructor TmbDBNavigator.Create(AOwner: TComponent);
var
  I: TNavigateBtn;
begin
  inherited Create(AOwner);
  for I := Low(Buttons) to High(Buttons)
    do Buttons[I].OnClick := Click;
end;
procedure TmbDBNavigator.Click(Sender: TObject);
begin
  BtnClick (TNavButton (Sender).Index);
end;
```

Pointer Notation

Most users are aware of Delphi's improved object instance notation, without the hat (^) operator. What Borland forgot to tell you is that in Delphi 2 this notation is supported for *all* pointer types that point to structured types (ie records, arrays etc). Check out Listing 4.

Contributed by Hallvard Vassbotn, hallvard@falcon.no

➤ *Listing 3*

```
{(C) 1996 Pijl Computer Services Ltd. Check laptop
power situation}
Unit Power;
Interface
Function PowerMan:Integer;
Implementation
Function PowerMan:Integer;
Var
  Fault,Batt,Life:Byte;
begin
  asm
    mov fault,00h
    mov ax,5300h
    mov bx,0000h
    int 15h
    jc @err
    mov ax,530ah
    mov bx,0001h
    int 15h
    mov batt,bl
    mov life,cl
    jc @err
    jmp @done
    @err: mov fault,AH
    @done: nop
  end;
  If (Fault=0) aAnd (Life in [0..100]) then
    PowerMan := Life
  else
    PowerMan := -1;
end;
end.
```

➤ *Listing 4*

```
type
  PMyRecord = ^TMyRecord;
  TMyRecord = record
    Field : longint;
  end;
  PMyArray = ^TMyArray;
  TMyArray = array[0..100-1] of longint;
  PMySimpleType = ^TMySimpleType;
  TMySimpleType = Double;
var
  tMyRec : TMyRecord;
  pMyRec : PMyRecord;
  tMyArr : TMyArray;
  pMyArr : PMyArray;
  tMySim : TMySimpleType;
  pMySim : PMySimpleType;

procedure NoHat;
begin
  { Initialize pointer variables }
  pMyRec := @tMyRec;
  pMyArr := @tMyArr;
  pMySim := @tMySim;
  { Test normal assignment }
  pMyRec^.Field := 1234567;
  pMyArr^[0]    := 1234567;
  pMySim^       := 3.14;
  { Test the no-hat assignment }
  pMyRec.Field := 1234567;  { Compiles fine! }
  pMyArr[0]    := 1234567;  { Compiles fine! }
  { This last one does not compile and should not }
  pMySim       := 3.14;
end;
```

Undocumented Delphi 2 Type Syntax

There are a lot of new language features in the Object Pascal versions found in Delphi 1 and Delphi 2, but there is one seemingly undocumented feature that escaped me for some time. In Delphi 2, the following constuct is legal:

```
type
  MyType = type longint;
```

This is used in the RTL/VCL in several places. For instance, `TDateTime` is defined as

```
TDateTime = type Double;
```

as can be seen from the on-line help or in the source (SYSUTILS.PAS). But what differentiates it from a normal `TDateTime = Double;` definition? One idea I had was that this would be a way to hide the underlying type and make the type checking of Object Pascal even stronger. To test this I tried all variations of assignments between variables with and without this extra `type` keyword. Everything compiled fine with no type errors – no luck there.

Then I tried sending the variables as `var` parameters to a procedure and, what do you know, the compiler gave me an *Error (54): Types of actual and formal var parameters must be identical*. Without the extra `type` keyword it compiles fine (see Listing 5).

So this gives us a way of defining our own types based on existing types, but giving better type checking when sending the type as `var` parameters. Good work, Borland! (if only you could document it as well...).

Contributed by Hallvard Vassbotn, hallvard@falcon.no

Delphi 2 Optimisation

I've been looking at how well the Delphi 2 optimiser generates code for typical `case` constructs. I built various versions of a simple `case` statement and used Turbo Debugger for Windows 32-bit (from TASM) to look at the resulting code. I must say I am totally awed by the designer and writer of the optimizer. It seems to have a mind of its own and is much less mechanical in its code generation than we are used to in Pascal products from Borland. Well done! See the file OPTCASE.PAS on this month's disk for the test details.

So the moral of the tale is that the optimiser will most probably generate very efficient code for your `case` constructs. You don't have to worry about sorting the entries or small gaps in the constants, the optimiser will still use very fast jump tables. If you are very concered with speed and have large gaps in the constants you could help the compiler by saying, for example:

```
if a = 100 then
  a:= 1
else
case a of
  1: a:= 1;
  2: a:= 1;
  ...
  17: a:= 1;
  18: a:= 1; { All of this will become a jump table }
end;
```

Contributed by Hallvard Vassbotn, hallvard@falcon.no

➤ *Listing 5*

```
type
  TStrong = type Double;
  TWeak   = Double;
procedure CheckWeak(var Strong: TWeak); begin end;
procedure CheckStrong(var Strong: TStrong); begin end;
procedure CheckDouble(var D: Double); begin end;
var
  D: Double;
  S: TStrong;
  W: TWeak;
begin
  CheckDouble(D); { compiles fine }
  CheckDouble(W); { compiles fine }
  CheckDouble(S); { <- compile error }
  CheckWeak(D);   { compiles fine }
  CheckWeak(W);   { compiles fine }
  CheckWeak(S);   { <- compile error }
  CheckStrong(S); { compiles fine }
  CheckStrong(D); { <- compile error }
  CheckStrong(W); { <- compile error }
end.
```

Your Delphi 2 Executables Are 500kb Larger Than You Think!

If you create a very small application in Delphi 2, then examine it using a utility such as Norton's System Information, you will be in for a shock! On a simple wallpaper-changing program brought to my attention recently, Norton reported that this 30Kb EXE file was actually 1.5Mb in size because of the various DLLs it referenced.

It turns out that as well as the normal references to USER32, KERNEL32 and so on, this tiny app pulled in the 500Kb OLEAUT32.DLL! It appears that the `SysUtils` unit uses some code to handle variants which references this DLL. You can verify it yourself with this sample program:

```
program HELLO;
uses Windows, SysUtils;
begin
  MessageBox(
    0, 'Hello world', 'Hello world', MB_OK );
end.
```

No calls whatever are made to `SysUtils`, but if you remove if from the `uses` clause the 500Kb OLEAUT32.DLL disappears from the project too!

In the `initialization` part of `SysUtils`, the exception handling logic is set up. This sets a global variable `ExceptionClass` to the daddy of all exception classes, `Exception`. Referencing this class links in all the code for `Exception`. Several methods in `Exception` call the `Format` routine, which calls `FmtStr` which in turn calls the `FormatBuf` routine, which is written in assembly. The `FormatBuf` routine checks the types of the arguments

sent in the array of const parameter. If the argument is a Variant type, it calls FormatVarToStr.

The code for FormatVarToStr simply contains an assignment from a Variant type to a longstring type. The compiler translates this into a call to VarToLStr, contained in the System unit. VarToLStr calls VarCast to convert the variant to a string.

VarCast indirectly contains calls to OLE2 calls such as VariantChangeTypeEx, VariantCopyInd, VariantClear, SysStringLen, SysAllocStringLen. The API calls are implemented in OLEAUT32.DLL which is imported in the System unit.

Now, this is clearly not a very good thing, so how could Borland fix it? Well, the simplest and least obtrusive way is to modify the behaviour of SYSTEM.PAS so that it does not link in these routines implicitly, by declaring them as external, but rather use procedure variables and only assign them and thus load the DLL when they are actually used for the first time, using the standard LoadLibrary and GetProcAddress APIs.

Ideally we that have the RTL/VCL source code should be able to do this ourselves, but I fear that it is not easy. First you need TASM 4.0 or later. Then you have to make the entire SYS directory by running the make file. Note that SYSTEM.PAS is compiled with an undocumented switch -y (this is probably because it contains some hard coded stuff that the compiler knows about). When (if) you have successfully re-compiled System and the other RTL units that depend on it, you have to re-compile the VCL. And as we all know not all of the VCL source has been distributed...

I haven't tried yet, but it might be that a modification of System is possible without re-compilation of anything else. The reason for this is that the OLEAUT32 routines are defined in the implementation section of the unit and any modifications we make there should not really affect the linking to other units.

Contributed by Hallvard Vassbotn, hallvard@falcon.no

Are You There?

I found a problem in the function FileExists in Delphi 1.0. In one of my forms I have a field to enter a filename and I want to check the existence of this file. But in a test I entered (incorrectly) '*.*' for the file mask and the FileExists function returned True! Maybe it's not an error but it could be a problem.

A better way to test the entry is using the FindFirst function. Look for it in my example program on the disk in CHKFILE.ZIP, the function I now use is:

```
Function ExistFile(FileName : String) : Boolean;
var
  SR : TSearchRec;
begin
  Result :=
    (FindFirst(FileName, faAnyFile, SR) = 0) and
    (SR.Name = FileName);
end;
```

Contributed by Peter Schrade, 101366,2433

TPageControl Accelerators

The standard Delphi 2 TPageControl doesn't support accelerator keys for its tabs, even when an accelerator is specified in the tab's caption. The unit in Listing 6 defines a TPageControl descendant that adds accelerator support by responding to the CM_DIALOGCHAR message. Note that the caption for each tab is actually stored in the page (TTabSheet) and that we should also check that the tab for a page is visible before checking its accelerator key. The Change method must be explicitly performed as it is not automatically called when the ActivePage property is changed.

Contributed by Dean Thompson, Classic Software, CompuServe 100033,1230

➤ *Listing 6*

```
unit NewPC;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, ComCtrls;
type
  TNewPageControl = class(TPageControl)
    private
      procedure CMDialogChar(
        var Message: TCMDialogChar);
        message CM_DIALOGCHAR;
    protected
    public
    published
  end;
procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TNewPageControl]);
end;

procedure TNewPageControl.CMDialogChar(
  var Message: TCMDialogChar);
var I: Integer;
    S: String;
begin
  if Enabled then
    with Message do begin
      for I := 0 to PageCount - 1 do begin
        S := Pages[I].Caption;
        if IsAccel(CharCode, S) and
          Pages[I].TabVisible then begin
          { select the appropriate Tab and
            give it focus }
          Result := 1;  { accelerator key is
                          valid (don't beep) }
          ActivePage := Pages[I];
          if ActivePage = Pages[I] then
            { successfully changed pages }
            Change;
          Exit;
        end;
      end;
    end;
  inherited;
end;

end.
```

Thanks for all your Tips – keep them coming in! If you have any hints that you think will be of use to fellow Delphi developers, just drop them in an email to the Editor on 70630.717@compuserve.com