Tips & Tricks

Edited by Mike Orriss. Please send your Tips direct to Mike at mjo@compuserve.com

Win32 API error codes

When making calls to Win32 API functions, most of them return an error code or set `GetLastError` should they fail. The error code returned is a 32 bit value. The error text associated with this 32 bit value can be found by a call to `FormatMessage` as shown below. The function below could added to your units or forms and used as follows:

```
errorSt:=GetLastErrorText(GetLastError);
```

Note the `DWORD` type is defined within Windows.pas as an `integer` (Listing 1).

Contributed by Andy McFarlane, andymac@clara.net

Two 'Gotchas'

I came across a couple of things that stumped me for an hour or so, and I thought they might be useful 'tips' for others. I wanted to use the Windows API call:

```
SetParent(child: HWnd; NewParent: HWnd)
```

but the compiler kept thinking I wanted to use the `TControl` method procedure:

```
SetParent(AParent: TWinControl); virtual;
```

The way to specify the Windows one is (sensibly enough) to refer to it as `Windows.SetParent`.

I was trying to use the `EnumChildWindows` procedure, but kept getting weird results to my callback function.

➤ *Listing 1*

```
function TForm1.GetLastErrorText(dwError:DWORD):string;
const
  MAX_MSG_SIZE = 256;
var
  szMsgBuf:array[0..MAX_MSG_SIZE-1] of char;
function MakeLangID(p, s:DWORD):DWORD;
begin
  result:= ((WORD(s) shl 10) or word(p));
end; {MakeLangID}
begin
  if FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, nil,
    dwError, MAKELANGID(LANG_ENGLISH,SUBLANG_ENGLISH_US),
    szMsgBuf, MAX_MSG_SIZE,nil) = 0 then begin
    result:=Format('Error %d',[dwError]);
  end else begin
    result:=szMsgBuf;
  end;
end; {GetLastErrorText}
```

It turned out (of course) to be that the definition of the callback I was using:

```
function EnumChildProc(childwnd : HWnd;
  lParam : LPARAM): Bool;
```

was incorrect. It needs to be:

```
function EnumChildProc(childwnd : HWnd;
  lParam : LPARAM): Bool; Stdcall;
```

so that the parameters get passed correctly. Yes, it says it plainly in the help file, but it's easy enough to forget.

Contributed by Mat Newman, Mat@and.co.uk

Update: Starting Projects Via DPR

As an alternative to Mark Erbaugh's suggestion to create a shortcut for every Delphi 1.0 project on a machine supporting multiple versions of Delphi (Issue 26), you could try this.

Open any Explorer window, go to `View|Options` and select the `File Types` tab. Select the `Delphi Project File` item and click `Edit`, then `New`, and enter `Open with Delphi 1.0` in the `Action` text box. Then navigate to your 16-bit Delphi executable (usually, this is `C:\DELPHI\BIN\DELPHI.EXE`) and click `OK`.

This will have the effect of adding an extra item to the Windows 95 right-click menu for any DPR file. The process can also be repeated for Delphi 2's executable should you have Delphi 3 installed.

Contributed by Russ Garner, rgarner@ondemand.co.uk

Update: Navigation With Cursor Keys (Issue 26)

It's true that there is no `WM_PREVDLGCTL` message, but one isn't needed as the first parameter (`wCtlFocus`) indicates if you want the `NEXT` control (`value=0`) or the `PREVIOUS` one (`value=1`).

Contributed by Lucas Franzen, luc@twc.de

Const Parameters

I am indebted to Mike Scott (mikes@compuserve.com) for the idea for this tip (which resulted from a pub conversation).

In Delphi 1 we all got used to passing string parameters as `Const` since it was more efficient to pass a pointer on the stack rather than the string itself. With Delphi 2 and 3 and the new string support, the strings are already passed as pointers so this reason is no longer valid. However, using `Const` is still more efficient since it prevents unnecessary reference counting. Mike has discovered (although it does not appear to be documented anywhere) that the same holds true when passing `Interfaces` as parameters.

Contributed by Mike Orriss, mjo@compuserve.com

Colours Property Editor

Listing 2 demonstrates how to add colour values to your system. Once registered, the additional values `clSproog` etc will be available for all `TColor` properties.

Contributed by Mike Scott, mikes@compuserve.com

FilteredComponent Property Editor

This is a property editor that shows how to have a custom property of the same type as the component itself, but not list the component in its own property dropdown list, ie so that the component can't refer to itself (Listing 3). Note the use of the abstract base class to work around a bug in Delphi 2.

Contributed by Mike Scott, mikes@compuserve.com

DbiDoRestructure Wrapper

This tip implements a wrapper around the Borland Database Engine's `DbiDoRestructure` function, which allows restructuring of non-SQL tables. However, why should we spend our time in understanding how to use that obscure routine when the Database Desktop does the job for us?

I've found a good reason for this during the development of my last project: I wanted the user to be able to modify at runtime the structure of a Paradox table, adding, deleting, renaming or moving fields. I could use the `ALTER TABLE` SQL statement in order to add and

➤ *Below: Listing 2*

```
unit CustomColours;
interface
uses
  SysUtils, Classes, Graphics, DsgnIntf ;
const
  clSproog  = $00234567 ;
  clFangle  = $00765432 ;
  clRelmist = $00767676 ;
  clPogarth = $0012AA21 ;
  Colors : array[ 0..3 ] of integer =
    ( clSproog, clFangle, clRelmist, clPogarth);
  ColorStrings : array[Low(Colors)..High(Colors)] of string
    = ('clSproog', 'clFangle', 'clRelmist', 'clPogarth');
type
  TMyColorProperty = class( TColorProperty )
  protected
    function  GetValue : string ; override ;
    procedure GetValues( Proc : TGetStrProc ) ; override ;
    procedure SetValue( const Value : string ) ; override ;
  end ;
// replacements for color functions in Graphics.pas
function ColorToString( Color : TColor ) : string ;
function StringToColor( const S : string ) : TColor ;
function ColorToIdent(Color : longint; var Ident : string) :
  boolean;
function IdentToColor(const Ident : string;
   var Color : longint) : Boolean;
procedure Register;
implementation
function ColorToIdent(Color : longint; var Ident : string) :
  boolean;
var i : integer ;
begin
  for i := Low( Colors ) to High( Colors ) do
    if Color = Colors[ i ] then begin
      Ident := ColorStrings[ i ] ;
      Result := true ;
      exit ;
    end ;
  Result := Graphics.ColorToIdent( Color, Ident ) ;
end;
function IdentToColor(const Ident : string;
  var Color : longint) : Boolean;
var i : integer;
begin
  for i := Low( ColorStrings ) to High( ColorStrings ) do
    if AnsiCompareText(Ident,ColorStrings[i]) = 0 then begin
      Color := Colors[i];
      Result := true ;
      exit ;
    end ;
  Result := Graphics.IdentToColor( Ident, Color ) ;
end ;
function ColorToString( Color : TColor ) : string;
begin
  if not ColorToIdent( Color, Result ) then
    FmtStr( Result, '$%.8x', [ Color ] );
end;
function StringToColor( const S : string ) : TColor ;
begin
  if not IdentToColor( S, Longint( Result ) ) then
    Result := TColor( StrToInt( S ) ) ;
end;
function  TMyColorProperty.GetValue : string ;
begin
  Result := ColorToString( TColor( GetOrdValue ) ) ;
end;
procedure TMyColorProperty.GetValues(Proc : TGetStrProc);
var
  i : integer ;
begin
  // add our colours first, call inherited first to have
  // our colours at the end of the list
  for i := Low( ColorStrings ) to High( ColorStrings ) do
    Proc( ColorStrings[ i ] ) ;
  inherited GetValues( Proc ) ;
end ;
procedure TMyColorProperty.SetValue(const Value : string);
var
  ColorValue : longint ;
  i : integer ;
begin
  if IdentToColor(Value, ColorValue) then
    SetOrdValue(ColorValue)
  else
    inherited SetValue( Value ) ;
end;
procedure Register ;
begin
  RegisterPropertyEditor(
    TypeInfo(TColor), NIL, '', TMyColorProperty);
end;
end.
```

➤ *Right: Listing 3*

```
unit FilteredComponentProperty;
interface
uses Classes, DsgnIntf ;
type
  TTestCompAbs = class( TComponent ) ;
  TTestComp = class( TTestCompAbs )
  private
    FParentComp : TTestCompAbs ;
  published
    property ParentComp : TTestCompAbs read FParentComp
      write FParentComp ;
  end ;
  TFilteredComponentProperty = class(TComponentProperty)
  private
    FInheritedProc : TGetStrProc ;
    procedure OurProc( const s : string ) ;
    function  IsOurName( const s : string ) : boolean ;
  protected
    procedure GetValues(Proc : TGetStrProc); override;
  end;
procedure Register;
implementation
function  TFilteredComponentProperty.IsOurName(
  const s : string): boolean;
var i : integer;
begin
  Result := true ;
  for i := 0 to PropCount-1 do
    if s = (GetComponent(i) as TComponent).Name then exit;
  Result := false ;
end ;
procedure TFilteredComponentProperty.OurProc(
 const s : string);
begin
  // only pass on if value is not our own name
  if not IsOurName(s) then FInheritedProc(s);
end ;
procedure TFilteredComponentProperty.GetValues(
  Proc : TGetStrProc);
begin
  FInheritedProc := Proc;
  inherited GetValues(OurProc);
end;
procedure Register;
begin
  RegisterComponents('Samples', [TTestComp]);
  RegisterPropertyEditor(TypeInfo(TTestCompAbs),
    TTestCompAbs, '', TFilteredComponentProperty);
end;
```

drop fields, but what about the remaining operations? I had no choice, I had to deal with the DbiDoRestructure function. Unfortunately, the documentation about that routine is quite poor: apart from the BDE Api help, I found only an example of use in Mike Orriss' DTopics archive. So, I had to do some experimentation myself.

The DbiDoRestructure function takes several parameters, as you can see examinating the DbiProcs.int file in Delphi 1 or the BDE API Help in Delphi 2; however only a few are necessary for my purpose, so I set the hDb parameter to my table's database handle, iTblDescCount to 1 (can't be otherwise), bAnalyzeOnly to False (I want to effectively restructure the table), and pszSaveAs, pszKeyViolName and pszProblemsName to NIL (I'm not interested for now).

The remaining parameter is the problematic one. The pTblDesc variable,of type pCRTblDesc, is a pointer to a complex record which holds the complete description of the table's structure. However, because I'm not interested in indexing, setting referential integrity, validation checks, etc, I can concentrate on only a few of the record fields: szTblName and szTblType (the table's name and driver type respectively), bPack (whether we want to pack the table also or not), iFldCount (the number of field descriptors I must provide), pFldDesc (the array of field descriptors) and pecrFldOp (array of crOpType structure, defining for each field the type of operation performed: add,modify, etc).

Most of the unit's code is devoted to the aim of properly filling these arrays, before calling the DbiDoRestructure function. First of all, the unit defines the enumerated type TResOp, which corresponds to the BDE crOpType, simply changing the cr prefix to res (eg crAdd becomes resAdd), to avoid conflicts. The crCopy operation is not used in the unit (frankly I don't understand it), I added instead a resMove constant to my TResOp type, that can be specified to change the position of a field in the table.

The Restructure procedure does the job of modifing the table structure. It takes several parameters, but only the first three are needed for all the restructuring operations: the table to restructure, the operation

➤ *Listing 4*

```
unit DBRestr;
interface
uses
  DB,DBTables,
  {$IFDEF Win32} Bde {$ELSE} DbiTypes, DbiProcs {$ENDIF};
type
  TResOp = (resADD,resDROP,resMODIFY,resMOVE);
  procedure Restructure(ATable:TTable; OpType: TResOp;
    FNum, FDest: integer; FName: string; FType: TFieldType;
    FSize: word);
implementation
uses
  SysUtils,
  {$IFDEF Win32}DBRUtl32 {$ELSE}DBRUtl16 {$ENDIF};
procedure Restructure(ATable:TTable; OpType: TResOP;
  FNum,FDest: integer; FName: string; FType: TFieldType;
  FSize: word);
type
  TFldArr = array[1..1000] of FldDesc;
  TOpArr = array[1..1000] of CROpType;
var
  hDb: hDbiDb;
  TblDesc: CRTblDesc;
  Dir:array[0..255] of char;
  pFldArr : ^TFldArr;
  pOpArr : ^TOpArr;
  FldCount,NewCount,j : integer;
  SaveActive: boolean;
  FDesc: FldDesc;
  Props: CURPROPS;
  TableTypeName : PChar;
begin
  with ATable do begin
    if Database.IsSqlBased then
      raise Exception.Create(
        'Cannot restructure SQL tables');
    SaveActive:=Active;
    if not Active then Active := true;
  end;
  TableTypeName := GetTableTypeName(ATable);
  Check(DbiGetDirectory(ATable.DBHandle, False, Dir));
  Check(DbiGetCursorProps(ATable.Handle, Props));
  FldCount := Props.iFields;
  if OpType = resAdd then
    NewCount := FldCount+1
  else
    NewCount := FldCount;
  if NewCount =0 then exit;
  pFldArr := AllocMem(NewCount * SizeOf(FLDDesc));
  pOpArr := AllocMem(NewCount * SizeOf(CROpType));
  Check(DbiGetFieldDescs(ATable.Handle, @pfldArr^[1]));
  try
    FillChar(TblDesc, sizeof(CRTblDesc), #0);
    TblDesc.bPack := True;
    case OpType of
      resModify :
        begin
          TblDesc.iFldCount := FldCount;
          with pFldArr^[FNum+1] do
            AnsiToNative(ATable.Locale, FName, szName,
              SizeOf(szName) - 1);
          pOpArr^[FNum+1]:=crModify;
          for j:=1 to TblDesc.iFldCount do
            pFldArr^[j].iFldNum := j;
        end;
      resAdd:
        begin
          TblDesc.iFldCount := FldCount+1;
          if FNum < FldCount then
            System.Move(pFldArr^[FNum+1],pFldArr^[FNum+2],
              (FldCount-FNum)*Sizeof(FldDesc));
            MapField(ATable, pFldArr^[FNum+1], FName,
              FType, FSize);
          pOpArr^[FNum+1]:=crAdd;
          for j:=1 to FNum do
            pFldArr^[j].iFldNum := j;
          if FNum < FldCount then
            for j:=FNum+2 to FldCount+1 do
              pFldArr^[j].iFldNum := j-1;
        end;
      resDrop:
        begin
          TblDesc.iFldCount := FldCount-1;
          if FNum < FldCount-1 then
            System.Move(pFldArr^[FNum+2],pFldArr^[FNum+1],
              (FldCount-FNum-1)*Sizeof(FldDesc));
          for j:=1 to FNum do
            pFldArr^[j].iFldNum := j;
          for j:=FNum+1 to FldCount-1 do
            pFldArr^[j].iFldNum := j+1;
        end;
      resMove:
        begin
          TblDesc.iFldCount := FldCount;
          for j:=1 to TblDesc.iFldCount do
            pFldArr^[j].iFldNum := j;
          FDesc := pFldArr^[FNum+1];
          if FDest > FNum then
            System.Move(pFldArr^[FNum+2],pFldArr^[FNum+1],
              (FDest-FNum)*Sizeof(FldDesc))
          else
            System.Move(pFldArr^[FDest+1],pFldArr^[FDest+2],
              (FNum-FDest)*Sizeof(FldDesc));
          pFldArr^[FDest+1]:=FDesc;
        end;
    end;
    ATable.Close;
    Check(DbiOpenDatabase(nil, nil, dbiReadWrite,
      dbiOpenExcl, nil, 0, nil, nil, hDb));
    Check(DbiSetDirectory(hDb, Dir));
    TblDesc.pFldDesc := @pFldArr^[1];
    TblDesc.pecrFldOp := @pOpArr^[1];
    if TableTypeName <> nil then
      StrCopy(TblDesc.szTblType, TableTypeName);
    StrPCopy(TblDesc.szTblName, ATable.TableName);
    Check(DbiDoRestructure(hDb, 1, @TblDesc, nil, nil, nil,
      False));
  finally
    Check(DbiCloseDatabase(hDb));
    FreeMem(pFldArr, NewCount * SizeOf(FLDDesc));
    FreeMem(pOpArr, NewCount * SizeOf(CROpType));
    if SaveActive then
      ATable.Open;
  end;
end;
end.
```

requested and the field number. To handle more easily the pFldDesc and pecrFldOp arrays, the procedure declares two Pascal style array types: TFldArr and TOpArr and two variables pointing to these types respectively: pFldArr and pOpArr.

After allocating the two array variables, I retrieve the original field descriptors in pFldArr, using the DbiGetFieldDescs API, then set to True the bpack parameter (it's a personal choice, of course) and then go into the case statement to select the proper operation.

If I must modify the name of the field (resModify) simply set the new name via the AnsiToNative procedure. Notice that I add one to the FNum parameter, because for Delphi the first field is zero, while for the BDE it is one. Adding a field is more complex: first, I must increase by one the Table Descriptor's iFieldNum field. Then, if the field is not appended to the end, I shift the field descriptors to make room for the new field. The MapField function *[Taken from the* DbUtl16 / DbUtl32 *units, whose code resembles too closely the Borland code to make it public, note: these compiled units are for Delphi 1 and Delphi 2 only, Mike Orriss]* retrieves the correct field descriptor for the new field and finally I reorder the iFldNum field of the field descriptors. In the remaining cases, analogous actions are performed, so the source code inspection should suffice to understand.

The Restructure procedure then closes the table, opens the table's Database in exclusive mode, makes the Table descriptor's pFldDesc and pecrFldOp fields point to our arrays, sets the szTblName and szTblType fields and finally calls the DbiDoRestructure function passing the address of our table descriptor. In the finally section (of the try..finally block), the database is closed, the allocated memory freed and the table is re-opened, if it was open before restructuring.

As you can see, the unit contains conditional compiler directives, in order to be compiled in either 16- or 32-bit mode. Depending on the Delphi version, it uses the DBRUtl16.dcu or the DBRUtl32.dcu files, whose source code I can't distribute for the reason above explained.

On this month's disk you will find the file DBRESTR.ZIP that contains a demo application DBRTest which is a test program for the DBRestr unit. It opens a Paradox table, DBRestr.db, located in the program directory, which initially contains a string field and an integer one. Pressing the Restructure button opens another form with a listbox containing the table's field names. The menu items allow adding (to the end), inserting, renaming or deleting fields. In the first two cases, a third form is displayed to request name and type of the new field (in this example only 30-chars strings and integers are allowed). The changes are immediately displayed in the list box, as well as in the main form's DBGrid. Moving fields is also supported by dragging a listbox item to the desired position. In the listbox DragDrop event I force the selection of the choosen item by simulating a mouse click via the Perform method (see the Brian Long's item about dragging on a DBGrid in the Issue 5 *Clinic*). Finally, the listbox contents are refreshed, in a BeginUpdate ... EndUpdate block, in order to make all changes in one step (Listing 4).

There are still things to do... For my present needs, the Restructure procedure has enough features, however a more comprehensive version could include referential integrity, validation checks, re-indexing, etc. But for now I have had enough restructuring . Maybe someday...

Contributed by Roberto De Marini,
rdemari@poboxes.com