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 Express Problem

Q I am using the InterBase Express components and am trying to control the login to a database. It works fine if the user uses the login dialog. It also works fine without the login dialog if the user name and password are set up in the `Params` property of the `TIBDatabase` component and the `LoginPrompt` property is set to `False`.

The problem I have arises when I make an `OnLogin` event handler and leave `LoginPrompt` set to `True` in order to trigger it. The event handler picks the user name and password from an INI file (this is not a high-security database) and sets the appropriate values in the `LoginParams` `TStrings` parameter as per the help instructions. However, despite this, I get an error indicating the user name and password are incorrect (Figure 1). How can I work around this error programmatically?

A The problem is caused by a simple logic bug in the IBX (InterBase Express) database component. I reported this problem when I bumped into it shortly after the release of Delphi 5; however, the problem was not fixed in Delphi 5's Update Pack 1, nor in C++Builder 5. At this stage I cannot predict if it will be fixed in Delphi 6.

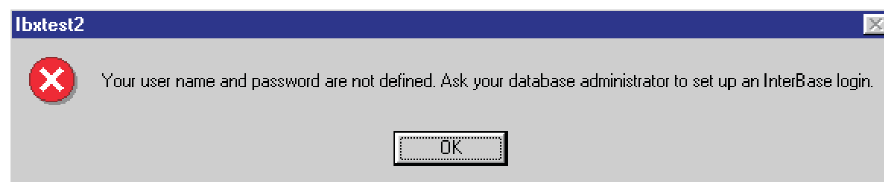
When a connection is required, the `TIBDatabase` component calls its `DoConnect` method. If `LoginPrompt` is `True`, `DoConnect` calls `Login` which calls the `OnLogin` event handler if present, otherwise it invokes the database login dialog. Assuming it called the `OnLogin` event handler, it then scans the database parameters `TStrings` object to see if it can find the password and remove it. This is done in a nested procedure called `HidePassword`, whose job is to remove the password entry and store it in a private string field called `FHiddenPassword`.

This job is, to be honest, done very poorly by `HidePassword` on two counts. Firstly, it checks each parameter string to see if it contains the substring `password` using `Pos`. However, it does the check wrongly. Instead of checking that `Pos` returns a non-zero value to indicate that `password` is found, it instead checks for a zero return. This means that the first non-password string it checks will be considered to be the password parameter.

As if that problem in itself was not enough, the code is supposed to then delete the password parameter line: however, the code avoids doing this if the password is found in the first parameter. Listing 1 shows this badly constructed routine, with comments added indicating what is wrong. Hopefully you can see the folly in the current implementation.

With this knowledge of why the routine fails you can work around the problem as follows. Make sure that the first parameter added is a `password` parameter, but with any arbitrary string as a value (possibly even the real password value). The next parameter added should use an arbitrary parameter name, but assign the real password to it. This way, the first item found not to contain the word `password` will be considered to have the actual password in it, and so the password will be assigned to `FHiddenPassword`. Then, since it was not the first parameter found, it will be deleted from the parameter list.

A sample pair of projects on the companion disk show the issue at work. IBXTest.dpr logs into the sample InterBase employee.gdb database thanks to the parameters set up in the Database Editor. This works just fine (as long as you make sure the path to the database



➤ *Figure 1: The unexpected IBX error.*

➤ *Listing 1: Dodgy code in IBX.*

```
procedure HidePassword;
var
  I: Integer;
  IndexAt: Integer;
begin
  //IndexAt should be initialised to -1 so that the default
  //value differs from any possible index position in Params
  IndexAt := 0;
  for I := 0 to Params.Count -1 do
    //Pos returns a non-zero value if it finds the target string,
    //not zero, so the comparison operator must be changed
    if Pos('password', LowerCase(Params.Names[i])) = 0 then begin
      FHiddenPassword := Params.Values[Params.Names[i]];
      IndexAt := I;
      break;
    end;
  //Here IndexAt is checked to see if a password was found. This should
  //check that IndexAt does not equal -1 (the changed initial value)
  if IndexAt <> 0 then
    Params.Delete(IndexAt);
end;
```

has been set up correctly in the form's OnCreate event handler). IBXTest2.dpr tries much the same thing but with an OnLogin event handler. You will see this fail if you set up the database path and run it. Listing 2 shows how some conditional compilation avoids the problem until it gets fixed. Just define the conditional symbol shown and the step discussed above allows the program to run and open a connection to the database.

As we go to press I've been alerted to an IBX patch at

http://www.interbase.com/open/downloads/IBX_updates.html

The patch appears to remedy this problem, but do note that the IBX update file is called IBXDP5EBETAUP41.EXE, which suggests (to me) that the patch is not quite finished yet.

Strings To Numbers

Q I have an application which translates many strings to integers and floating points, for which I use StrToFloat and StrToInt. When these routines are given invalid input they raise an exception. This is fine in the norm, but is a pain when debugging in the IDE as the debugger keeps popping up. Also it presumably has a performance overhead with many translations which I would prefer to do without. Is there a way of translating strings to numbers without getting exceptions?

A The debugger can be told not to intercept application exceptions, either by disabling it completely or by just turning off the exception interception feature (see Table 1 for information on where these options are in the various incarnations of Delphi). Delphi 4 introduced more debugger features including the ability to ignore certain types of exceptions. In Tools | Debugger Options... you can use the Add... button to add additional exception classes over and above EAbort for the debugger to ignore, such as EConvertError.

The alternative solution is to do what the normal translation routines do and avoid generating the exception. This is quite easy given that we have the RTL source code provided with Delphi Professional and Enterprise (or Client/Server Suite, as it used to be called).

StrToInt is implemented with a call to the Val procedure. If an error code is returned by Val, StrToInt raises an exception. StrToFloat is implemented with a call to TextToFloat, raising an exception if it returns False. With this information to hand we can implement our own versions of the routines that return Boolean values instead of raising exceptions.

Possible implementations can be seen in Listing 3. Since the functions return Boolean values, the translated numbers are returned in var parameters. Notice that TextToFloat took only two parameters in Delphi 1, as the Currency type was not introduced until Delphi 2 (when TextToFloat was extended to cater for both Extended and Currency values).

Typed Constants

Q It seems to be possible to declare a persistent variable that is only available to the

Listing 2: Working around the IBX login bug.

```
{$define CateringForIBXLoginBug}
procedure TForm1.IBDatabase1Login(Database: TIBDatabase;
  LoginParams: TStrings);
begin
  LoginParams.Values['password'] := 'masterke';
 {$ifdef CateringForIBXLoginBug}
  LoginParams.Values['xassword'] := 'masterke';
 {$endif}
  LoginParams.Values['user_name'] := 'sysdba'
end;
```

Listing 3: Translation routines without exceptions.

```
function StringToInt(const S: string; var I: Integer): Boolean;
var
  E: Integer;
begin
  Val(S, I, E);
  Result := E = 0
end;
{$ifdef Win32}
function StringToFloat(const S: string; var E: Extended): Boolean;
begin
  Result := TextToFloat(PChar(S), E, fvExtended)
end;
{$else}
function StringToFloat(const S: string; var E: Extended): Boolean;
var
  Buf: array[0..255] of Char;
begin
  Result := TextToFloat(StrPCopy(Buf, S), E)
end;
{$endif}
```

Table 1: Locating debugger options in Delphi.

Version	Integrated Debugger	Exception Interception
Delphi 1	Options | Environment... | Preferences | Integrated debugging	Options | Environment... | Preferences | Break on exception
Delphi 2	Tools | Options... | Integrated debugging	Tools | Options... | Break on exception
Delphi 3	Tools | Environment Options... | Integrated debugging	Tools | Environment Options... | Break on exception
Delphi 4 and 5	Tools | Debugger options... | Integrated debugging	Tools | Debugger options... | Language Exceptions | Stop on Delphi Exceptions

```
type
  TDataRec = packed record
    Text: String;
    Number: Double;
  end;
const
  Captions: array[Boolean] of String = ('Disabled', 'Enabled');
  DataRec: TDataRec = (Text: 'Hello world'; Number: Pi);
```

➤ *Listing 4: Some typed constants.*

```
function GetCount: Integer;
const
  Count: Integer = 0;
begin
  Inc(Count);
  Result := Count;
end;
```

➤ *Listing 5: Using a typed constant as a static variable.*

subroutine within which it is declared. I came across this by accident and can't decide whether it is a bug or a feature of Delphi. It could be useful for debugging.

A What you are referring to are *typed constants*. These contrast to the normal (untyped) constants that most Delphi developers will be used to. Those normal constants are useful for defining identifiers used to represent simple values. In cases where you want to have more interesting values predefined at compile-time and associated with an identifier, typed constants are very helpful. They allow you to set up initialised records, arrays and pointers.

Listing 4 shows two typed constants. One is an array of strings, indexed by the values `False` and `True`, and the other is an example of a record. Both array elements and both record data fields have been initialised with values.

This is the original intent of typed constants, to help set up pre-initialised non-simple type values. However, a side effect of their implementation is that storage is allocated by the compiler to hold these values. Additionally, this storage is global as opposed to local, which means it is allocated at program start and exists through the lifetime of the program.

This last point would be irrelevant were it not for the key point that typed constants are not (by default) constant. Instead, Borland made them writeable. This makes them act like initialised variables. However, normal variables (defined in `var` sections) can only be initialised if they are non-local, meaning defined outside any subroutine. This is because the compiler only supports initialising global data, which it does by storing initialised data values in the EXE itself. So initialised variables are pre-initialised in the executable, not by any code that executes.

The fact that typed constants can be defined as either local to a subroutine or globally therefore needs to be examined carefully. Since the compiler does *not* support initialising items by auto-generated code, it follows that typed constants are initialised in the same way as global variables, by data being stored in the executable. Typed constants are initialised at program startup and at no other time.

So, even if a typed constant is defined to have local accessibility, it has global storage and is initialised at startup. The implication of this is that any change made to any typed constant (global or local) in any subroutine will be persistent. The next time the subroutine is invoked, the typed constant will still have the changed value. Typed constants are the equivalent of *static variables* in C. Listing 5 shows a simple function using a typed constant to return an incrementing counter. It returns 1 the first time it is called, 2 the second time, 3 the third time and so on.

Clearly the term *typed constant* is a misnomer, albeit one that has persisted through Delphi's Turbo Pascal and Borland Pascal heritage. In fact, using typed constants as an equivalent to C static variables was covered on the first page of my old Borland Pascal problem solving book from 1994.

However, Delphi 1 users took offence at the possibility of something called a constant being modified, so Delphi 2 introduced a compiler switch to fix the loophole. You can toggle a switch on the `Compiler` page of the project options dialog to deny programmers the ability to write to typed constants. This switch corresponds to the `$J` or `$WriteableConst` compiler directives.

`{$J+}` or `{$WriteableConst On}` permits typed constants to be modified (this is the default, for backward compatibility with Delphi 1 and earlier compilers, despite what Delphi 2's help claims). `{$J-}` or `{$WriteableConst Off}` makes typed constants truly constant.

Editing List View Items

Q `TListView` controls look quite useful but appear baffling to use, to me anyway. One thing I can't figure out how to do is allow the user to edit the items in the list view. I have tried various combinations of properties and it resolutely refuses to be edited.

A Strangely, despite your efforts (or possibly in spite of them), the Windows list view control deals with this all by itself. Like many controls it supports a number of user-driven operations (mouse and keyboard) in order to support editing.

The *Windows 98 Resource Kit* documents `F2` as being a shortcut for a rename operation. Editing a list view item is considered a rename at the user interface level. Take the Windows desktop for example. If you select any icon there and press `F2`, you can rename it. This causes an in-place editor to appear, allowing you to modify the text associated with the list view item.

In actual fact, you are not really renaming anything. In component terms, you are actually changing the list view item's `Caption` property. Anyhow, without quibbling over terminology, `F2` does what you need. Additionally, as you probably have often found in Windows Explorer to your annoyance, if a list view item is already

selected a single-click will also allow the caption to be edited (see Figure 2).

So, from scratch, you can click on an item, then pause, then click again to invoke the edit operation. The pause is necessary to ensure that the list view does not take the two individual clicks as a double-click.

The programmatic equivalent of either `F2` or click-pause-click is to call the `TListItem` object's `EditCaption` method. `EditCaption` ensures the owning list view has focus and then sends a `LVM_EDITLABEL` message to it, specifying the list item's `Index` property.

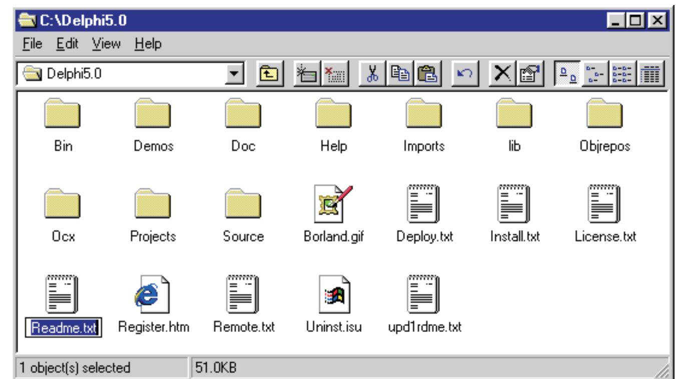
Date Comments

Q I have a need to be able to stick a quick comment in my Delphi code, showing what's been changed today. I use a Code Template to do it. I have a code snippet called *day* and pressing `Ctrl+J` on it causes something like `{ 23/3/00-JB }` to be inserted. The only disadvantage is having to update the date in the editor options every day, but that's far outweighed by the usefulness. Is there a better way?

A Code Templates are very handy for that sort of comment, but only when the comment will not change very often. You set new Code Templates up on the `Code Insight` page of the `Tools | Editor Options...` (Delphi 5 and later) or the `Tools | Environment Options...` (Delphi 3 and 4) dialogs (Figure 3 shows an example). For readers who are unfamiliar with Code Templates, I should mention that they were introduced in Delphi 3 and are inserted with `Ctrl+J`, which gives a list of all available templates. You can also get a cut-down list by entering the first letter/s of the template name before pressing `Ctrl+J`.

If you need a Code Template whose definition varies, Delphi does not really help you, but it *is* possible to help yourself here.

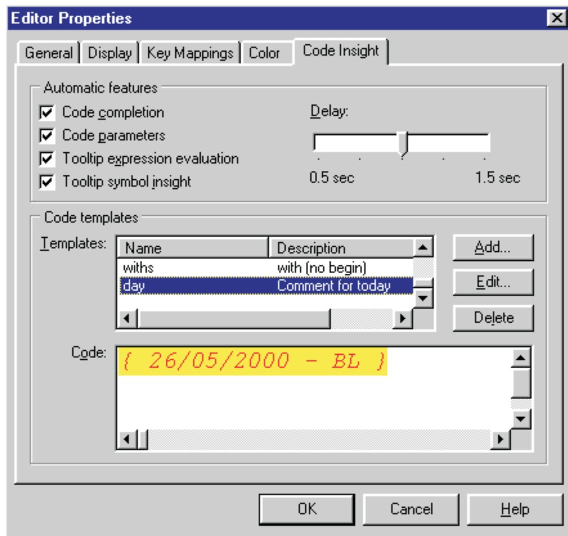
You could write a small utility program whose job is to update



➤ *Figure 2: Editing an item's caption in Windows Explorer.*

➤ *Listing 6: Programmatically updating Code Templates.*

```
program UpdateDCI;
uses
  SysUtils, Registry, Windows, Classes, ShellAPI,
  Dialogs, Controls;
const
  DelphiRegPath = 'Software\Borland\Delphi';
  Delphi3 = '3.0';
  Delphi4 = '4.0';
  Delphi5 = '5.0';
  CodeTemplateFileName = '\Bin\Delphi32.dci';
  DelphiExe = '\Bin\Delphi32.exe';
  //Update these constants as appropriate
  //Your Code Template as stored in the DCI file
  //(Name | Description)
  Template = 'day | Comment for today';
  //Your version of Delphi
  MyDelphi = Delphi5;
  //Your desired comment string, with a placeholder for the
  //date
  Comment = '%s - BL';
  //Your preferred date format
  DateFormat = 'd/mm/yyy';
  //Where Delphi should start, relative to its root.
  //Make sure you prefix with a \
  DelphiStartDir = '\Projects';
  //Custom command-line parameters for Delphi
  DelphiStartParams = '/np /hm /hv';
var
  DelphiRoot: String;
  CodeTemplates: String;
  TemplateStart, TemplateEnd: Integer;
  Reg: TRegIniFile;
  TemplateFile: TStream;
procedure RunDelphi;
begin
  ShellExecute(0, nil, PChar(DelphiRoot + DelphiExe),
    DelphiStartParams, PChar(DelphiRoot + DelphiStartDir),
    SW_SHOWNORMAL)
end;
begin
  try //Application exception handler
    Reg := TRegIniFile.Create('');
    try
      Reg.RootKey := HKEY_LOCAL_MACHINE;
      if not Reg.OpenKey(DelphiRegPath, False) then
        raise Exception.Create(
          'Delphi information not found.');
      DelphiRoot := Reg.ReadString(MyDelphi, 'RootDir', '');
      if DelphiRoot = '' then
        raise Exception.Create(
          'Delphi root path not found.')
    finally
      Reg.Free
    end;
    //Read DCI file into a string
    TemplateFile := TFileStream.Create(DelphiRoot +
      CodeTemplateFileName, fmOpenRead or fmShareDenyWrite);
    try
      SetLength(CodeTemplates, TemplateFile.Size);
      TemplateFile.Read(CodeTemplates[1], TemplateFile.Size)
    finally
      TemplateFile.Free
    end;
    //Locate our template
    TemplateStart := Pos(Template, CodeTemplates);
    if TemplateStart = 0 then
      raise Exception.Create(
        'Custom Code Template not found.');
    //Now locate the comment
    while CodeTemplates[TemplateStart] <> '{' do
      Inc(TemplateStart);
    TemplateEnd := TemplateStart;
    while CodeTemplates[TemplateEnd] <> '}' do
      Inc(TemplateEnd);
    //Substitute old date for today's date
    Delete(CodeTemplates, TemplateStart, TemplateEnd -
      TemplateStart + 1);
    Insert(Format('{ ' + Comment + ' }',
      [FormatDateTime(DateFormat, Date)]),
      CodeTemplates, TemplateStart);
    //Write DCI string back to file
    TemplateFile := TFileStream.Create(
      DelphiRoot + '\' + CodeTemplateFileName, fmCreate);
    try
      TemplateFile.Write(CodeTemplates[1],
        Length(CodeTemplates));
    finally
      TemplateFile.Free
    end
  except
    //If there is a problem, check it's ok to launch Delphi,
    //otherwise leave
    on E: Exception do
      if MessageDlg(
        Format('%s'#13#13'Continue loading Delphi',
          [E.Message]), mtError, [mbOk, mbCancel], 0) =
          mrCancel then
          Exit
  end;
  //Launch Delphi now template has been updated
  RunDelphi
end.
```



➤ *Figure 3: Defining a custom Code Template.*

the Code Template to have today's date in. The utility can then spawn Delphi. If you set your Delphi shortcut to point to the small utility application, the whole operation will then become transparent.

Naturally, you need to know how to find the Code Template before writing the utility. You will find them all in Delphi's BIN directory in the file DELPHI32.DCI. This file contains only text, and so some reasonably straightforward text parsing is all you need.

Listing 6 shows a simple program that does the job. It is intended to be modified before use and you should see a number of constants that can be changed as appropriate.

`Template` represents the name and description of your Code Template as specified when you defined it. It is used to locate the template within the file and so should be set up carefully, as detailed in the comment. `MyDelphi` should be set to refer to the appropriate constant for your Delphi version, either `Delphi3`, `Delphi4` or `Delphi5` (currently).

`Comment` represents the commented text you want to insert where the *%s* placeholder will be replaced by the date, formatted as described in `DateFormat`. Since the application is designed to launch Delphi you can specify which directory (relevant to the Delphi root) should be made current with `DelphiStartDir`. This must have a backslash prefix and defaults to Delphi's `Projects` directory, but you may have a different preference.

Finally, `DelphiStartParams` can be set to an empty string or `nil`, but you can also use it to specify command line parameters to Delphi. The default parameters specified in the listing cause Delphi to start with no default project open (`/np`), and enable both heap monitoring (`/hm`) and heap verification (`/hv`).

Persistent Run-Time Column Data

Q I have an application that uses a number of `TDBGrid` components where the users are at liberty to reorder the columns and change their widths. What is the easiest way of storing this information when the program closes so I can restore the information when the program restarts?

A Since Delphi 3, the object represented by the `Columns` property of a `TDBGrid` has had useful methods for storing and retrieving all the column attributes (see Listing 7). If you do not mind having several files you could call the `Columns.SaveToFile` method for each `TDBGrid` in the application. However, if you want all the information to be stored in one file, it may be necessary to use streams.

➤ *Listing 7: Useful methods for TDBGrid.Columns.*

```
TDBGridColumns = class(TCollection)
...
public
  procedure LoadFromFile(const Filename: string);
  procedure LoadFromStream(S: TStream);
  procedure SaveToFile(const Filename: string);
  procedure SaveToStream(S: TStream);
  ...
end;
```

➤ *Listing 8: Storing and retrieving column attributes.*

```
const
  ColumnData = 'COLUMNS.DAT';
procedure TForm1.LoadGridsFromStream(AOwner: TComponent);
var I: Integer;
begin
  for I := 0 to AOwner.ComponentCount - 1 do begin
    if AOwner.Components[I] is TDBGrid then
      TDBGrid(AOwner.Components[I]).Columns.LoadFromStream(
        Stream);
    LoadGridsFromStream(AOwner.Components[I])
  end
end;
procedure TForm1.SaveGridsToStream(AOwner: TComponent);
var I: Integer;
begin
  for I := 0 to AOwner.ComponentCount - 1 do begin
    if AOwner.Components[I] is TDBGrid then
      TDBGrid(AOwner.Components[I]).Columns.SaveToStream(
        Stream);
    SaveGridsToStream(AOwner.Components[I])
  end
end;
procedure TForm1.LoadColumnData;
begin
  if not FileExists(ColumnData) then
    Exit;
  Stream := TFileStream.Create(ColumnData, fmOpenRead
    or fmShareDenyWrite);
  try
    LoadGridsFromStream(Application)
  finally
    Stream.Free
  end
end;
procedure TForm1.SaveColumnData;
begin
  Stream := TFileStream.Create(ColumnData, fmCreate);
  try
    SaveGridsToStream(Application)
  finally
    Stream.Free
  end
end;
procedure TForm1.FormShow(Sender: TObject);
begin
  LoadColumnData
end;
procedure TForm1.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
  SaveColumnData
end;
```

My solution relies on all the grids existing as the program closes, and also all existing as the main form is first displayed. This is just to keep the code fairly short. You can cater for more complex cases by modifying the code as you like.

When the program closes, the `SaveColumnData` method is called (Listing 8). This method creates a file stream which in turn creates a column data file (overwriting one if it already exists). Whilst the new file is open `SaveGridsToStream` is called. The purpose of this recursive method is to locate all `TDBGrid` components by starting with the `Application` object and then using the ownership mechanism to hunt out forms and components on forms. Each grid has its column data saved to the stream before continuing the search.

When the program starts, `LoadColumnData` does much the same thing. It opens a file stream for the column data, if present, before calling `LoadGridsFromStream`. This attempts to locate all grid components one at a time, telling them to read their column data from the stream.

Hopefully this should give you some ideas as to how to achieve your goal. The code can be found on this month's disk as the project ColumnSaving.dpr.

Delphi Grammar Problem

Q I am using the `CreateProcess` API to unzip a file to a directory (see Listing 9) and the code waits for the process to finish using `WaitForSingleObject`. I am getting cases where the process loops continually and `WaitForSingleObject` does not force it to timeout. What might the problem be?

```
procedure TForm1.Button1Click(Sender: TObject);
var
  SysDir, CommandLine: String;
  SI: TStartupInfo;
  PI: TProcessInformation;
begin
  SetLength(SysDir, MAX_PATH);
  SetLength(SysDir, GetSystemDirectory(@SysDir[1], MAX_PATH));
  CommandLine := SysDir + '\Command.com /c UNZIP.exe Archive.zip';
  Win32Check(CreateProcess(nil, PChar(CommandLine), nil, nil, True, 0, nil,
    'c:\Data', SI, PI));
  while WaitForSingleObject(PI.hThread, 40000) <> (WAIT_OBJECT_0 or WAIT_TIMEOUT)
    do begin
    //nothing
  end;
  CloseHandle(PI.hProcess);
  CloseHandle(PI.hThread);
end;
```

➤ *Listing 9: Failing code.*

A I think the problem here is caused by improper expression construction. You have a `while` statement operating with this condition:

```
WaitForSingleObject(
  PI.hProcess, 40000) <>
  (WAIT_OBJECT_0 or
  WAIT_TIMEOUT)
```

This statement calls `WaitForSingleObject` and then checks the return value. You are intending to ensure the value equals neither `WAIT_OBJECT_0` nor `WAIT_TIMEOUT`, but that is not what you get.

The compiler will evaluate this `Boolean` expression as follows.

First, the API is called, which will return with a value. The compiler then tests to see whether this value differs from the value (`WAIT_OBJECT_0 or WAIT_TIMEOUT`). If we replace the constants with their literal values, this means the compiler will be checking the return value does not equal (0 or $102). The `or` operation is interpreted as a bit-wise `or` and so the bracketed section evaluates to $102 (0 or $102 equals $102).

This means that your loop will continue calling `WaitForSingleObject` whilst it does not return $102, or `WAIT_TIMEOUT`. This is likely to carry on for a long time, assuming the original unzip operation takes less than 40 seconds to execute. Each time it is called, `WaitForSingleObject` is likely to return `WAIT_OBJECT_0`, indicating that the process handle has become signalled because the process has terminated.

To fix the problem, change the code to look like either version in Listing 10. You can see two possibilities there, one of them being commented. The commented version takes advantage of both constants representing values less than 256 and so uses set notation. It verifies that the value returned by `WaitForSingleObject` is not a member of the set containing both `WAIT_OBJECT_0` and `WAIT_TIMEOUT`.

The other version that is not commented uses a more traditional approach. It stores the `WaitForSingleObject` result in a temporary variable and then explicitly checks if the value is `WAIT_OBJECT_0` or if it is `WAIT_TIMEOUT`.

[As an aside, there are a number of good Delphi components available for zipping and unzipping files. One of the best is VCLZip, from http://vclzip.bizland.com. Ed]

➤ *Listing 10: Waiting for a process to finish.*

```
procedure TForm1.Button1Click(Sender: TObject);
var
  SysDir, CommandLine: String;
  SI: TStartupInfo;
  PI: TProcessInformation;
  Res: DWord;
begin
  SetLength(SysDir, MAX_PATH);
  SetLength(SysDir, GetSystemDirectory(@SysDir[1], MAX_PATH));
  CommandLine := SysDir + '\Command.com /c UNZIP.exe Archive.zip';
  Win32Check(CreateProcess(nil, PChar(CommandLine), nil, nil, True, 0, nil,
    'c:\Data', SI, PI));
  //Ensure the return value is not in the specified set of values before going
  //again while not WaitForSingleObject(PI.hThread, 40000) in
  //  [WAIT_OBJECT_0, WAIT_TIMEOUT] do begin
  //  //nothing
  //end;
  //Ensure result is either one or the other before terminating the loop
  repeat
    Res := WaitForSingleObject(PI.hThread, 40000)
  until (Res = WAIT_OBJECT_0) or (Res = WAIT_TIMEOUT);
  CloseHandle(PI.hProcess);
  CloseHandle(PI.hThread);
end;
```

```
procedure TForm1.UpdateTableFromUI;
var
  Stream: TStream;
begin
  try
    tblBioLife.Edit;
    { Save edit control into string field }
    fldCommonName.AsString := edtCommonName.Text;
    Stream := TBlobStream.Create(fldNotes, bmWrite);
    try
    { Save memo control into BLOb field }
      memNotes.Lines.SaveToStream(Stream)
    finally
      Stream.Free
    end;
    tblBioLife.Post
  except
    tblBioLife.Cancel
  end
end;
procedure TForm1.UpdateUIFromTable;
var
  Stream: TStream;
begin
  { Disable buttons as appropriate }
  btnPrior.Enabled := not tblBioLife.Bof;
  btnNext.Enabled := not tblBioLife.Eof;
  { Load edit control from string field }
  edtCommonName.Text := fldCommonName.AsString;
  Stream := TBlobStream.Create(fldNotes, bmRead);
  try
    { Load BLOb field into memo control }
    memNotes.Lines.LoadFromStream(Stream);
  finally
    Stream.Free
  end;
  { Reset change detector }
  FieldChanged := False
end;
procedure TForm1.MoveRecord(MoveBy: Integer);
begin
  { If change made, ask if it should be saved }
  if FieldChanged and
    (MessageDlg('Record has changed. Update table?',
    mtConfirmation, [mbYes, mbNo], 0) = mrYes) then
    UpdateTableFromUI;
  { Go to requested record }
  tblBioLife.MoveBy(MoveBy);
  { Refresh UI }
  UpdateUIFromTable
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
  tblBioLife.Open;
  UpdateUIFromTable
end;
procedure TForm1.btnPriorClick(Sender: TObject);
begin
  MoveRecord(-1)
end;
procedure TForm1.btnNextClick(Sender: TObject);
begin
  MoveRecord(1)
end;
{ Event handler shared by edit's and memo's OnChange events }
procedure TForm1.FieldChange(Sender: TObject);
begin
  { When edit/memo is changed, set change detector }
  FieldChanged := True
end;
```

➤ *Listing 11: Accessing BLOb fields with a TBlobStream.*

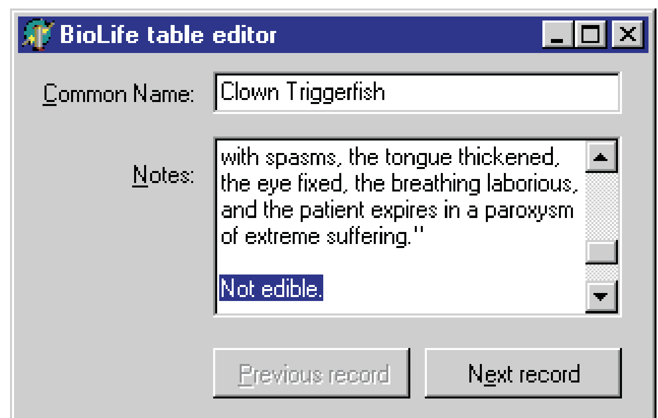
Memos And BLOb Fields

Q I have set up a text memo field in a Paradox table. How do I get data from a memo control into this field and vice versa?

A A memo field is an example of a BLOb field. A BLOb is a **B**inary **L**arge **Ob**ject, for example an arbitrary amount of text, an image or any arbitrary binary data. Because of how the acronym originated, I am pedantic in spelling it *BLOb*, as opposed to *BLOB* or *Blob* which are common alternatives *[Glad someone is upholding correct standards around here! Ed]*.

When you access a database table using standard components with Delphi, you will need to take special care with BLOb fields. Normal fields pose no problem, as you can easily use a dataset's `FieldValues` property (an array of `Variant` values, indexed by the field name) or alternatively the various access properties of a field object (such as `AsString`, `AsInteger` and so on).

BLOb fields, however, need to be accessed using a special kind of stream called a `TBlobStream`. To show the idea, a program called BLObRead.dpr is on this month's disk that uses normal controls to access a couple of fields from the sample BioLife.db Paradox table. The program works in any version of Delphi.



➤ *Figure 4: Reading and writing a BLOb field (and a string field).*

A `TTable` component is used to access the table and has persistent field objects manufactured with the Fields Editor to simplify access to the `Common Name` and `Notes` fields. `Notes` is the BLOb field, by the way. Both persistent field objects have been given a name prefixed with `fld` for easy identification. You can see the program running in Figure 4.

All the application code is shown in Listing 11. The two key routines are `UpdateTableFromUI` and `UpdateUIFromTable`. You can see the `TBlobStream` object being created in both routines, one with a `bmWrite` mode and one with a `bmRead` mode. Fortunately, liiasing between the stream and the memo is easy thanks to the `TStrings` class defining the `LoadFromStream` and `SaveToStream` methods.

The rest of the code helps the basic application user interface work, so I will let you peruse it at your leisure.

Acknowledgements

Thanks are due this month to Jack Birrell who alerted me to the existence of the column saving/loading methods within `TDBGrid`.