The Delphi CLINIC

Edited by Brian Long

Problems with your Delphi project? Just email Brian Long, our Delphi Clinic Editor, on clinic@blong.com

TWebBrowser Documentation Problem

Q I'm trying to figure out how to get (read and write) the data returned by the `PostData` parameter of the `TWebBrowser.BeforeNavigate2` event. It's supposed to be a `Variant` array, but no matter what I try, I can't seem to get a sensible value out of it (or in it).

```
<HTML>
<BODY>
<H1>
Data entry:
</H1>
<FORM ACTION="/scripts/WebTest.exe/ShowDetails" METHOD="POST">
Enter your name:
<INPUT TYPE=TEXT NAME=YourName>
<P>
<INPUT TYPE=SUBMIT VALUE="Submit">
</FORM>
</BODY>
</HTML>
```

➤ *Listing 1: The HTML input form from the web server application.*

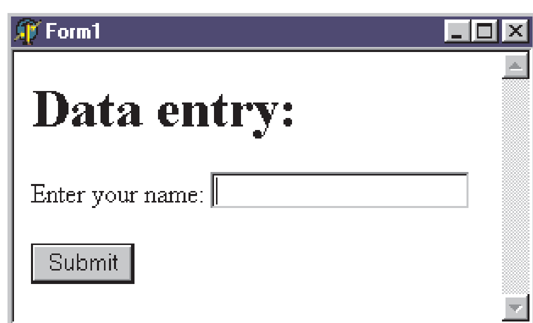
A Before looking for the answer to this question, we should understand what the questioner is asking. `TWebBrowser` is the ActiveX version of Internet Explorer. I described how to import and install this ActiveX onto the ActiveX page of the component palette of Delphi 2, 3 and 4 in Issue 47 (the *Using Internet Explorer* entry in *The Delphi Clinic*). Delphi 5 comes with the component pre-installed on the component palette's `Internet` page.

The component allows you to write an application that has web browsing facilities. Its `BeforeNavigate2` event allows you to intercept any URL that the browser is told to go to, whether by program control, by user input, or by hyperlinks on a web page. The job of the `PostData` parameter of the event handler is to represent data sent to the server when an HTTP post operation occurs, for example by an HTML input form. `PostData` should contain the unparsed string giving the values to be posted with their associated fields.

If you are familiar with writing web server applications using Delphi's Web Bridge technology (applications that are started with `File | New... | Web Server Application`) then `PostData` represents what will ultimately be the `Content` property of a `TWebRequest` object. In a web module, accessing the `Request` property gives you a `TWebRequest` object.

Now we can start looking at the problem itself. A sample web browsing application is on this month's disk as WebBrowser.Dpr. This is a Delphi 5 project using a `TWebBrowser` component, but which would be easy to change to work with Delphi 3 or 4. The primary issue is that when the Internet Explorer ActiveX is imported in these earlier versions, the event handler parameters are a little different.

➤ *Figure 1: The web server application's input form.*



The program navigates to a specific URL. The URL represents a call to a CGI web server application, which is also on the disk as WebTest.Dpr. The web browser project assumes WebTest.Exe will be found in the Scripts directory off the web server's virtual root directory. This means that it also assumes your machine has some web server running on it (I was running Microsoft's Personal Web Server to test this problem).

The techniques and issues of writing web server applications are beyond the scope of this column, but you can find many articles on the subject in back issues of *The Delphi Magazine*. The *Collection '99* CD-ROM, containing the first 48 issues, should help you out here.

WebTest.Exe is a web server application (which compiles in Delphi 3 or later) that can display two pages of HTML. The default page is an input form that asks the user for a piece of information (their name) which is then passed back to the server via the `Submit` button. The HTML for this is shown in Listing 1 and you can see the application running in Figure 1. This shows that when the `Submit` button is pressed, WebTest.Exe is run again with a `/ShowDetails` path, and the text typed into the edit box on the form, which represents the `YourName` field, will be posted back alongside.

When the web browser application launches the input form the `BeforeNavigate2` event will fire, but there will be no posted fields and so `PostData` is not expected to contain anything useful. When the application user types in their name and presses the `Submit` button the event will fire again.

This time, PostData should contain all the details of posted fields. The question is asking how to get this information from the parameter.

If you read the documentation for the event, you will see that PostData is a var parameter (a pass by reference parameter) of type OleVariant, which is a more true-to-Windows version of a Variant. You should be able to examine this parameter with the various Variant support routines in the System unit to ascertain what type of data it holds. You might expect the data to be some form of string, or maybe a Variant array of bytes, given the varying length of textual data that should be there and this, I think, is where the querstioner's problem lies.

If you pass PostData to the VarType function, it will return a value made up of bit flag constants, defined in the System unit. A statement in the event handler like this:

```
ShowMessageFmt(
  'Data type is $%x',
  [VarType(PostData)]);
```

tells us the type data is $400C, which is equivalent to the expression varVariant or varByRef. What this means is that PostData does not actually hold the data we are interested in. It instead holds a reference to another Variant that holds the data. Before you can start examining what data was posted, we have to extract this other Variant.

To do this, you can declare a local OleVariant variable in the event handler and use code like Listing 1. With this code, Tmp will now contain the posted field data. To find in what format the data is stored, you can make a call to VarType(Tmp), which returns $2011, equivalent to the expression varArray or varByte. This proves that ultimately the data is stored as a Variant array of bytes.

Since the data is textual, extra code can extract the data from the byte array and do anything with it. Listing 3 writes the information on the form's caption bar (see Figure 2) by locking the array and reading directly from its contents into a string, after calculating how long the string should be.

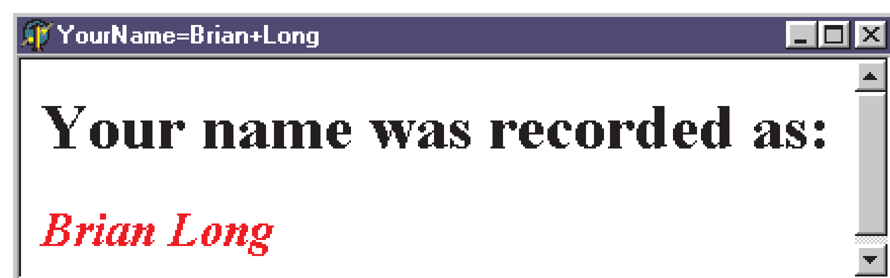
```
procedure TForm1.BrowserBeforeNavigate2(Sender: TObject;
  const pDisp: IDispatch; var URL, Flags, TargetFrameName, PostData,
  Headers: OleVariant; var Cancel: WordBool);
var
  Tmp: OleVariant;
begin
  Tmp := OleVariant(TVarData(PostData).VPointer^);
end;
```

➤ Above: Listing 2

➤ Below: Listing 3

```
procedure TForm1.BrowserBeforeNavigate2(Sender: TObject;
  const pDisp: IDispatch; var URL, Flags, TargetFrameName, PostData,
  Headers: OleVariant; var Cancel: WordBool);
var
  Tmp: OleVariant;
  S: String;
  P: Pointer;
begin
  Tmp := OleVariant(TVarData(PostData).VPointer^);
  if VarIsArray(Tmp) then begin
    SetLength(S, VarArrayHighBound(Tmp, 1) - VarArrayLowBound(Tmp, 1) + 1);
    P := VarArrayLock(Tmp);
    try
      Move(P^, S[1], Length(S));
      Caption := S
    finally
      VarArrayUnlock(Tmp)
    end
  end
end;
```

➤ Figure 2: The BeforeNavigate2 event handler at work.



Local Share Requirement

Q I read your article about *Paradox File Corruption* and was happy to find one place that covers so many aspects of this annoying problem. We are about to add the source code to our projects and would like to ask the following. Are all the Windows 95 issues also relevant to Windows 98? Is there a way to change the LOCAL SHARE to TRUE automatically? Have you come across any more information on the subject since writing the article?

A As far as I am aware, with the exception of the driver issue, which is fixed in Windows 98, all the registry settings are either relevant to Windows 98, or will do no harm if applied.

The code in Listing 4 sets LOCAL SHARE to TRUE in the BDE configuration file. You can see that it requires the BDE unit to be added to your uses clause and then makes a number of BDE API calls to do the job.

When the BDE Administrator is used to change configuration settings, it warns that all BDE applications must be closed and restarted for the change to take effect. The same is true with this code snippet.

After applying all the changes made in the original article, some people still seem to get issues here and there. However, the article represents all I know on the subject, assuming you are referring to the updated version on the web at www.itecuk.com/delmag/paradox.htm rather than the original article in Issue 42.

Computer Picker

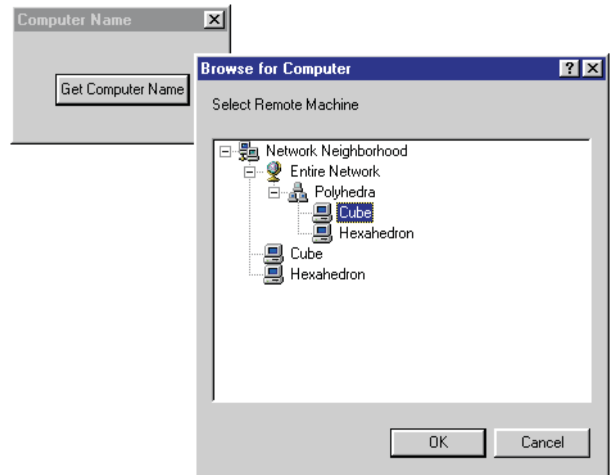
Q I am starting to write a client application for a DCOM

server. The server application may be located on a number of machines around the network, and I want the client application user to choose which machine to run it from. This means I need some way to choose a machine from the various ones on my network. How do I do this?

A Rather than trying to work this one out for myself, I thought I'd see what I could find within Delphi itself. The `TDCOMConnection` component in my copy of Delphi 5 has a property called `ComputerName`, which has a property editor that does exactly what you want. It launches a dialog showing the network and all the machines on it, allowing you to choose one. The `TSocketConnection` component has a `Host` property that does exactly the same.

The nice thing about Delphi 5 is that it comes with the source to many of the property and component editors used by the standard components. In the Source\Property Editors directory, the file MidReg.Pas contains the code for the `MIDAS` property and component editors. A class called `TComputerNameProperty` does just what we want. Well, almost. Of course, the class is designed as a property editor, so a small amount of work is required to turn it into a function (see Listing 5).

You can see the function makes use of a Windows shell routine, `SHBrowseForFolder`, set up to look for computer names. A simple test project is on the disk as ComputerNameSelector.Dpr and can be seen strutting its stuff in Figure 3.



➤ *Figure 3: Selecting a computer name.*

```
uses
  BDE;
procedure EnsureLocalShareIsTrue;
var
  Cursor: HDBICur;
  ConfigDesc: CFGDesc;
begin
  Session.Open;
  Check(DbiOpenCfgInfoList(
    nil, dbiREADWRITE, cfgPERSISTENT, '\SYSTEM\INIT', Cursor));
  try
    while DbiGetNextRecord(Cursor, dbiNOLOCK, @ConfigDesc, nil) = 0 do
      with ConfigDesc do begin
        if (CompareText(szNodeName, 'LOCAL SHARE') = 0) and
           (CompareText(szValue, 'FALSE') = 0) then begin
          szValue := 'TRUE';
          Check(DbiModifyRecord(Cursor, @ConfigDesc, True));
        end
      end
  finally
    DbiCloseCursor(Cursor)
  end;
end;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
uses
  ShlObj;
function GetComputerName: String;
var
  BrowseInfo: TBrowseInfo;
  ItemIDList: PItemIDList;
  ComputerName: array[0..MAX_PATH] of Char;
  WindowList: Pointer;
  Success: Boolean;
begin
  if Failed(SHGetSpecialFolderLocation(
    Application.Handle, CSIDL_NETWORK, ItemIDList)) then
    raise Exception.Create('Computer Name Dialog Not Supported');
  FillChar(BrowseInfo, SizeOf(BrowseInfo), 0);
  BrowseInfo.hwndOwner := Application.Handle;
  BrowseInfo.pidlRoot := ItemIDList;
  BrowseInfo.pszDisplayName := ComputerName;
  BrowseInfo.lpszTitle := 'Select Remote Machine';
  BrowseInfo.ulFlags := BIF_BROWSEFORCOMPUTER;
  WindowList := DisableTaskWindows(0);
  try
    Success := SHBrowseForFolder(BrowseInfo) <> nil;
  finally
    EnableTaskWindows(WindowList);
  end;
  if Success then
    Result := ComputerName
  else
    Result := ''
end;
```

Form Painting Puzzle

Q I do not understand why the button `OnClick` event handler (shown in Listing 6) does not work the way it is supposed to.

When I click the button, the application waits for one second and then the panel turns blue. It completely bypasses the first line. If I add a call to `Application.ProcessMessages` after the line that changes the panel to red, then everything works fine, but then I understand that calling `Application.ProcessMessages` repaints the screen.

Why does the panel not turn red of its own volition? I cannot reproduce this problem with Visual Basic 5.

A The answer to this question is that the panel does not turn red upon execution of the relevant statement because it is unable to do so at that point. In the case of a Visual Basic panel, I can only assume that the implementation has extra code to ensure that it does happen, taking a not entirely dissimilar approach to that of `Application.ProcessMessages`.

To appreciate the problem fully requires an understanding of how event handlers are executed in the

context of an application running in a Windows environment, and what exactly `Application.ProcessMessages` does.

Windows is a message-based system. When certain things happen, such as the user moving the mouse or pressing keys, or a timer's time-out expiring, messages are sent along to various windows in the system. Some other causes of messages include settings in Control Panel being changed, and an application's window(s) becoming invalid, possibly due to the application being restored from an icon, or being made visible thanks to another application being closed.

Typically, the messages are dropped into a per-application message queue. Your application has a message loop, tucked away inside the `Application` object, in the private `ProcessMessage` method (not to be confused with the public `ProcessMessages` method mentioned in the question). This loop polls the message queue to see if any messages are present. If there are, the first one is plucked out and code executes to process it.

What processing is required depends upon the message that is found. It may just require calling the Windows API that asks for standard message processing if the application is not interested in doing anything special (`DefWindowProc`). However, for many messages, the VCL will have message handlers set up to perform special processing. In many cases, these message handlers will call event handlers of components that the messages were targeted to.

When a message is present in the queue as a result of a button being pressed, the following sequence of events occurs at some later point. The message loop extracts the message from the queue and gives it, in some way, to the button. The button's appropriate message handler calls a routine inside the button (called `Click`) to check if there is an event handler set up for an `OnClick` event. If so, the event handler is called. The event handler executes, taking as long as it needs to do so and then finally exits. Control goes back to `Click`, then back to the message handler, and then finally back to the message loop which can then check for more messages.

The point here is to note that no messages are being plucked from the message queue whilst the event handler is executing. Consequently, if the event handler itself executes statements that will cause extra messages to be placed in the message queue, they will be left there until some point after the event handler has completed.

Now let's look at the event handler in the question. The first statement assigns a new colour to the panel. This requires the panel to be redrawn. In the standard Windows way, this is achieved by posting a `wm_Paint` message into the application message queue. This message will stay in the message queue until the message loop is next able to extract messages. The event handler then proceeds to suspend itself for one second, but the message loop is still inactive, waiting for both the event handler, and the message handler that called the event handler, to finish. The next action is for another colour to be given to the panel, causing another `wm_Paint` message to be placed in the message queue.

At this point, there are conceptually two `wm_Paint` messages sitting in the application message queue waiting to be processed. Neither of them has been processed at this point because the message loop is still waiting for the event handler and message handler to finish. At last, the event handler exits, the message handler can exit and the message loop then picks up the paint messages and deals with them. The net effect is that after all statements in the event handler have executed, the panel turns blue.

Clearly, because of the way the message loop in a Windows application operates, if several paint messages like the ones generated here for a given window (such as a control) are placed in the queue, only the last one has any real effect. Because of this, Windows actually optimises multiple paint messages by merging them all into one. So in truth, by the time the event handler terminates, there is only one paint message waiting to be processed. The statement that changed the panel to red ends up causing nothing to happen.

The whole point of the question is how to make sure the red colour assignment *does* have an effect. Which translates to a question of how to get messages pulled from the queue and processed when the main message loop is waiting for event handlers etc to finish.

The normal answer to this question is to suggest a call to `Application.ProcessMessages` somewhere after the statement that causes the message(s) to go into the queue (see Listing 7). `Application.ProcessMessages` executes another version of the message loop, which runs through the queue, processing all the messages it finds there. It does not necessarily cause the form to be redrawn, unless there happens to be a `wm_Paint` message pending the queue, targeted at the form window.

The thing about `Application.ProcessMessages`, though, is that it processes *all* messages, not just

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Panel1.clColor := clRed;
  Sleep(1000);
  Panel1.clColor := clBlue;
end;
```

➤ *Above: Listing 6*

➤ *Below: Listing 7*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Panel1.clColor := clRed;
  Application.ProcessMessages;
  Sleep(1000);
  Panel1.clColor := clBlue;
end;
```

the paint message. If you just want the panel to be turned red, and you do not want any other mouse clicks, drags, key presses and so on to be processed, then `Application.ProcessMessages` is overkill. Fortunately, all visual controls have a dedicated method that allows you to get any pending paint messages for them to be processed if there are any, but otherwise to do nothing. This method is called `Update`. Listing 8 shows the code using `Update`.

The questioner mentioned that Visual Basic didn't suffer from this 'problem'. I can only assume that when you change any property of a Visual Basic object that has an on-screen ramification, it calls the equivalent of `Update` automatically. An event handler that makes many modifications to the appearance of one object will therefore cause many screen updates in a Visual Basic application, but very few in a Delphi application. This may be one of the reasons that screen updating in Visual Basic applications is deemed to be slower than would be desirable to their users.

Dual Processor Woes

Q I have an application that has been running perfectly well on several machines for some time. Now, one of my customers has installed a dual processor machine, and this is where the problem starts. Every now and again my application fails in various ways, including Access Violations, on the dual processor machine. The problem is intermittent and unpredictable and I am wondering if there are known issues with Delphi applications on these type of machines.

A Assuming your application is multi-threaded and is written in Delphi 2, 3 or 4, then I may have some useful information for you. During the development of Delphi 5, a lot of work was put into isolating certain intermittent problems that did crop up with Delphi applications running on SMP (that is, **S**ymmetric **M**ulti-**P**rocessing) hardware.

Primarily the threading problem identified in Delphi 4 was with string reference counting. The RTL string handling routines read the string reference count into a CPU register, increment (or decrement) the register, then put the value back into memory. That's a problem if multiple threads are running around. If a thread switch occurs in the middle of this three-step cycle, and another thread does the same thing to the same string you'll end up with a heap leak (string not freed) or a double free (string freed twice, which is potential Access Violation territory).

Now, threads have been around in Delphi for ages, and strings never really presented a problem before. At least, not noticeably. To get this to be a problem, you had to be accessing the same string instance from multiple threads, like reading from a global string variable, or from a string property of a global or shared object. Even then, the window of opportunity for this bug to reproduce is very, very small (only three instructions wide). On a single processor machine, the time-slice between threads is relatively large (several milliseconds) so the chances of a thread switch happening right in the middle of those three instructions is very slim.

With multiple processors executing code simultaneously, the granularity of concurrency is reduced from the several millisecond time-slice down to a single instruction. Every instruction is an opportunity for a thread collision, because the other processor is running code at the same time. So, even though the window of opportunity is still only three instructions wide, we now have a 1,000 times greater chance of threads colliding in that area.

The test case Inprise used was an application that would sometimes raise an exception if left running for days at a time on a single processor machine. They were never able to get this to happen on demand in the debugger. When the application was run on a dual processor machine, the crash occurred in less than two minutes.

The fix for strings in Delphi 5 was to devise a way to do reference counting in a thread-safe manner. The simplest solution would be to use an application-wide critical section. Lock the critical section before accessing/updating the memory, and release it when done. That's fine for most code (critical sections are the least intrusive form of thread synchronisation), but string reference counting happens *a lot* in a Delphi application, so performance was still a worry.

The realisation came that since all the string RTL routines are written in assembler, and they only need to know if decrementing the reference count produces a zero value, they could replace the read/modify/write sequence of three instructions with a single inc/dec memory instruction. The inc/dec memory instruction will set the CPU flags register to indicate if the result was a zero or not.

Now, a single instruction to increment a value in memory is still a read/modify/write operation. It is not an atomic operation from the viewpoint of the system memory bus. To make it atomic (and multi-processor safe), they added a `LOCK` prefix. This locks the system bus for the duration of the instruction, preventing the other processor from seeing that area of memory until the instruction has completed.

`LOCK` should also force the other processor to flush its internal cache so that it does not read old cached values of that updated memory location. Smart chipsets will only invalidate the cache lines that refer to that memory area, whereas not so smart chipsets (like the PII) flush the entire cache. Flushing caches may be expensive, but it's still cheaper than a critical section.

➤ *Listing 8*

```
procedure TForm1.Button1Click(
  Sender:TObject);
begin
  Panel1.clColor := clRed;
  Panel1.Update;
  Sleep(1000);
  Panel1.clColor := clBlue;
end;
```

If you're working at the source code level, it is best to assume that all source code statements involve more than one machine operation.

Probably 90% of threading problems are caused by unprotected read/modify/write cycles. Multiple threads can *read* from a shared value unsynchronised without causing problems. The problems start when that shared value needs to be updated.

Look for global variables and other shared resources (properties of a global object) as possible thread collision points. Local variables and parameters, stored on the stack, are inherently thread-safe.

Pay attention to COM threading models. In particular, access to global data as well as instance data (implicitly accessed through `Self`) is not thread-safe in free-threaded apartment model (`Free` in the COM object wizard's `Threading Model:` option).

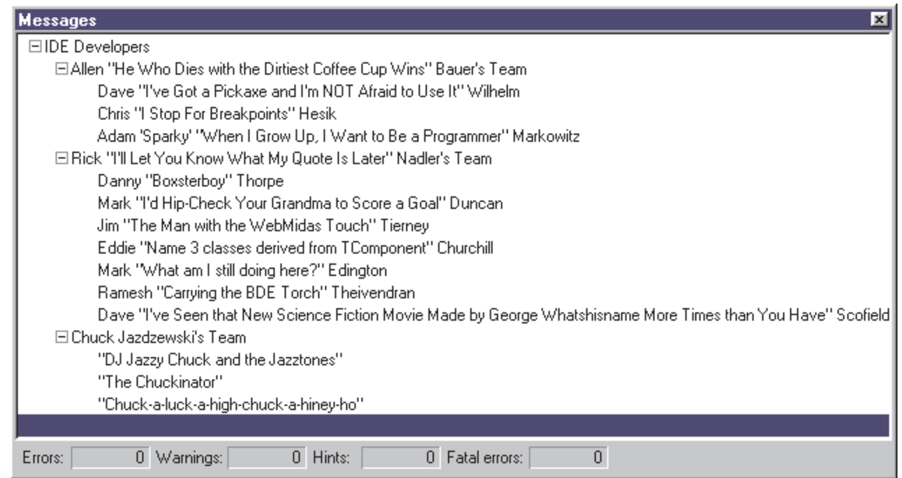
All the above information can be used to isolate why your application may be failing. If it is string access, then moving to Delphi 5 might be an option. If it is some other resource that has not been properly synchronised, then that can be fixed. An alternative to working out how to remedy the problem is to avoid the problem. Multi-processor machines allow applications to restrict themselves to executing on a single processor.

A call to `GetProcessAffinityMask` returns gives you a `DWord` containing a bit mask indicating which processors the threads of the application can run on. You can then modify the affinity mask and call `SetProcessAffinityMask` to restrict the application to a single processor.

What? More Easter Eggs?

As an update to the list of Delphi 5 Easter Eggs presented in Issue 50, here are details of another one.

Allen Bauer, the Delphi and C++Builder IDE R&D Manager and Senior R&D Engineer, was recently dropping hints in a newsgroup about an obscure Easter Egg that is accessed without going to the About box. He mentioned that it is



➤ *Figure 4: The obscure Delphi 5 Easter Egg uncovered.*

not well known internally at Inprise, let alone by people outside the company. Hallvard Vassbotn, writer of many low-level articles in past issues of this magazine, took the bait and set to work finding it. Here are the results.

Make a text file, either in the IDE with `File | New... | Text`, or using Notepad, or Windows Explorer. Save the text file with a `.Allen` extension, as opposed to a `.Txt` extension. Start a new project in Delphi 5 and open the project manager (`View | Project Manager` or `Ctrl+Alt+F11`). Right click on the project, choose `Add...` and use the `Files of type:` combobox to show `Any file (*.*)`. Choose the `.Allen` file and press `Open` to add it to the project.

At this point, right clicking on the text file will give you a menu including the entry `Meet Allen Bauer`. Choosing this menu item first displays a message box saying *Hi I'm Allen* and then launches your web browser to show the following URL:

```
www.on24.com/corporatevideo/
  borland/3-5_56.html
```

This is a web page showing a video of Allen Bauer talking about some of the new features added in C++Builder 4.

If you now compile the project (`Ctrl+F9`), the compiler will produce some additional credits to the IDE developers (see Figure 4). A side effect of the compilation is to also produce another empty text file with the same name, but a `.Allen.Bauer` extension.

Corrections

As can be expected in a problem-solving column, where many issues are dealt with each month, occasional errors inevitably creep in.

In Issue 50, the *Custom TLabel Component* entry described a hyperlink-style label component. Listing 10 showed code that would work in any version of Delphi, but the file on disk had older code, that will only work in 32-bit versions of Delphi. This month's disk has an updated version of the file, HLLabel.Pas that works just as well with version 1 as with any other version.

In Issue 49's *Daylight Savings Changeover* entry, the project made use of some constants that were only defined in the Windows unit in Delphi 4 and later, causing problems if you try and compile in Delphi 2 or 3. A new version of the project is on the disk this month which compiles happily in all 32-bit versions. Unfortunately this was not the only problem. As you can see in Figure 1 of p66 of Issue 49, the UK daylight savings change dates were out by a whole month *[Oops! Sorry]*. The updated code on this month's disk fixes that error as well.

Acknowledgements

Thanks go to Danny Thorpe from the Delphi R&D team for sharing his knowledge of SMP hardware issues and how Delphi 5 now deals with them better.