How to make a TWebBrowser become the active control when clicked

Introduction

When you have a form containing a *TWebBrowser* control and you click in the control it does not automatically become the active control of the parent form. However, tabbing around a form does make the web browser the active control. This behaviour is unlike other controls such as *TEdit* and *TMemo* and can cause some unexpected behaviour in your application, as I learned the hard way!

This article prevents two methods of ensuring that the *TWebBrowser* control behaves like other controls by making it the active control when clicked. As we will see, there are problems with either approach, but the second method is the most robust.

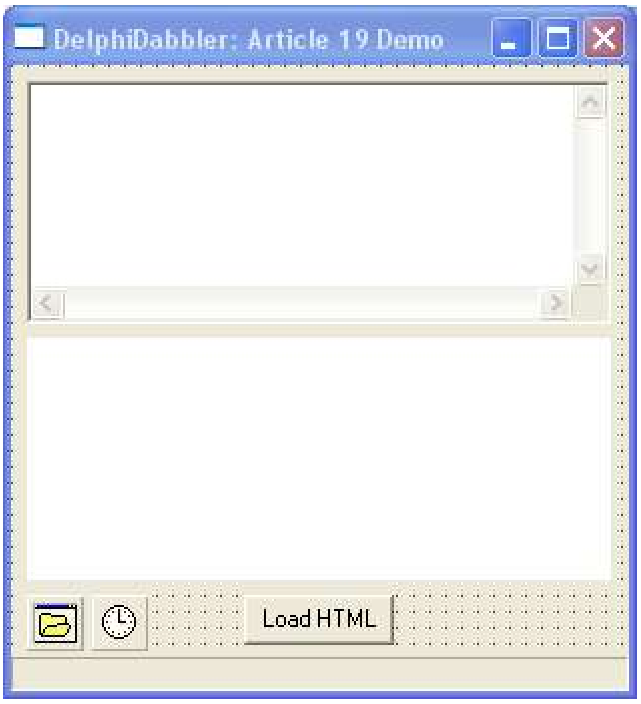
Investigate the problem

To see the problem let's create a small application that shows the name of the parent form's active control in the status bar.

Start a new Delphi application and drop the following controls on the form, keeping the default names:

- A *TMemo* – clear its *Lines* property and set its *TabOrder* property to `0`.
- A *TWebBrowser* – sets its *TabOrder* property to `1`.
- A *TButton* – set its *Caption* to `'Load HTML'` and its *TabOrder* property to `2`.
- A *TTimer* – set its *Interval* property to `100`.
- A *TOpenDialog* – set its *Filter* property to display files with `.html` and `.htm` extensions.
- A *TStatusBar*.

Arrange the form and set its *Caption* to look something like this:



We will use the *TTimer* to update the status bar with details of the active control every 1/10th of a second. To do this create a *OnTimer* event handler for the *TTimer* as follows:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if Assigned(ActiveControl) then
    StatusBar1.SimpleText := 'ActiveControl = '
```

```
      + ActiveControl.Name
  else
    StatusBar1.SimpleText := 'ActiveControl = nil';
end;
```

Listing 1

Compile and run the application and move around the controls using the TAB key. You will see the name of each control displayed in the status bar as it becomes active. To see the problem use the mouse and click in either the memo or edit control. The status bar will change to show its name. Now click in the web browser. Nothing will change.

Maybe this is because the web browser is not displaying any HTML – i.e. it has no document object. Let us fix this by adding the facility to display a HTML document. Double click the *TButton* to create an *OnClick* event handler and add the following code to it:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
    StatusBar1.SimpleText := '';  // fixes SB redraw problem
    if FileExists(OpenDialog1.FileName) then
    begin
      WebBrowser1.Navigate('file://' + OpenDialog1.FileName);
      Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
    end
    else
    begin
      WebBrowser1.Navigate('about:blank');
      Memo1.Text := 'about:blank';
    end;
  end;
end;
```

Listing 2

This method displays the open file dialog box. If the user OKs we check if the provided file exists. If so we display it in the web browser and show its source code in the memo control. If the file doesn't exist we navigate to `about:blank` and display that text in the memo.

Recompile the application and run it. Load a HTML file into the web browser (or create an `about:blank` document by entering an invalid file name). When you try clicking the various controls you will see that the problem persists. We have more work to do.

A solution

This solution lies in detecting a mouse click in the web browser control and, in reponse, setting the form's *ActiveControl* property to reference the web browser. In essence we wan't to do something like this:

```
// !!! Wrong !!!
procedure TForm1.WebBrowser1Click(Sender: TObject);
begin
  ActiveControl := Sender as TWebBrowser;
end;
```

Listing 3

Unfortunately, *TWebBrowser* does not have an *OnClick* or *OnMouseDown* event in which to do this, so we have to find an alternative way of detecting the mouse click.

The web browsers's *OnCommandStateChange* event is useful here. According to Delphi's help, this event is triggered when the ability to execute certain *TWebBrowser* methods changes. It happens that this event is triggered when, amongst other things, the user clicks the mouse in a document.

Note that I said "amongst other things". The *OnCommandStateChange* event is not only triggered when the mouse is clicked in the document – it is also triggered on other occasions. So we need to find when the event is in response to such a click. Research and experimentation shows that we can reliably detect a mouse click in the following circumstances:

- The event handler is called with a *Command* parameter of *CSC_UPDATECOMMANDS*.
- The document's *selection* object is available and has a type of 'Text'.

Now to use the *selection* object we must access the browser's document object – and to access the document object we must have loaded a document into the browser. This means we can't use this code on the web browser control until a document has been loaded. The simplest way to ensure this is by navigating to `about:blank`. We can do this in the Form's *OnShow* event handler which we add now:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  WebBrowser1.Navigate('about:blank');
end;
```

Listing 4

Having ensured there is a document present in the web browser we can use the following code in the browser's *OnCommandStateChange* event handler:

```
procedure TForm1.WebBrowser1CommandStateChange(Sender: TObject;
  Command: Integer; Enable: WordBool);
var
  Doc: IHTMLDocument2;        // document object
  Sel: IHTMLSelectionObject;  // current selection
begin
  // Check we have a valid web browser triggering this event
  if not Assigned(Sender) or not (Sender is TWebBrowser) then
    Exit;
  // Check we have required command
  if TOleEnum(Command) <> CSC_UPDATECOMMANDS then
    Exit;
  // Get ref to document object and check not nil
  Doc := WebBrowser1.Document as IHTMLDocument2;
  if not Assigned(Doc) then
    Exit;
  // Get ref to current selection
  Sel := Doc.selection as IHTMLSelectionObject;
  // If selection is of correct type then we have a mouse click
  if Assigned(Sel) and (Sel.type_ = 'Text') then
    // Make the web browser the form's active control
    ActiveControl := Sender as TWebBrowser;
end;
```

Listing 5

Here we first make sure that the sending object is a non-nil *TWebBrowser* control. We then check the *Command* parameter and exit if it is not *CSC_UPDATECOMMANDS*. Next we try to get hold of the browser's *Document* object, exiting if it is nil. From a valid document object we get the document's current selection object, again checking it is non-nil. We also check the selection type is `'Text'`. If these tests succeed we finally set the form's *ActiveControl* property to the web browser and we are done.

Once again, recompile the project and tab and click about the form. You should now find the web browser becomes the active control when clicked or tabbed into.

«« Gotcha #1 »»

There's a gotcha here. I carefully used the word "document" rather than "control" in the above description. The *OnCommandStateChange* event only detects clicks in the current document, not necessarily in the control. This is because the browser, by default, places a border round the document. Clicking in this border is not detected because it is **not** part of the document.

The only (partial) solution I've heard of is to make the border disappear using CSS. This is OK if you have access to the document – you can use something like:

```
body {margin: 0;}
```

The technique is no good if you can't change or override the document's CSS. We can improve matters by setting the browser's default CSS (See *article #18* for details of how to do this from code). However, if the document sets its own border it will override any default, so we're stuck.

Thanks to *ANBe* for pointing out the cause of this problem and suggesting the CSS fix. Anyone who has any other ideas please *get in touch*.

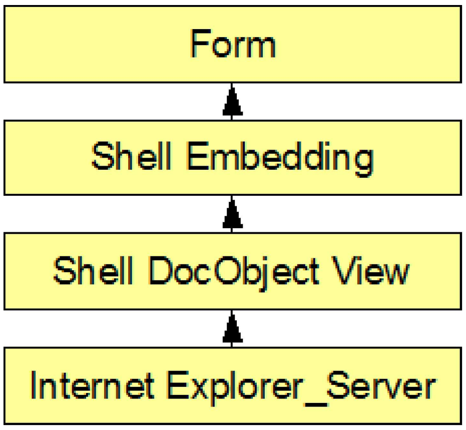
«« Gotcha #2 »»

There's more. The solution requires the document's selection type to be "text". Unfortunately, if you click on a non-text item in the document, for example and image, the selection type is not "text".

I'm afraid I don't have a solution to this problem as yet.

Another Solution

Bärje Henriksson has suggested another solution that does detect clicks on the whole control – and so avoids the "Gotchas" above.



Bärje's solution depends on finding the window handle for the browser control. We do this by searching for a window whose class is named "Internet Explorer_Server". This is ultimately a child of the form window. The diagram on the left shows the relationship.

No *TApplicationEvents*?

If your version of Delphi doesn't have the *TApplicationEvents* component all is not lost. Alternatives are discussed below.

And so to the code. Re-using the code we developed for the previous solution, add a private field named *IEServerWindow* of type *HWND* to the form class and drop a *TApplicationEvents* component on the form. Now remove the *WebBrowser1CommandStateChange* we created in *listing 5*.

Modify the timer component's OnTimer event handler as shown in *listing 6* below:

```
procedure TDemoForm.Timer1Timer(Sender: TObject);
var
  NextWin: HWND;  // handle of various child windows
begin
  while IEServerWindow = 0 do
  begin
    NextWin := FindWindowEx(DemoForm.Handle, 0, 'Shell Embedding', nil);
    NextWin := FindWindowEx(NextWin, 0, 'Shell DocObject View', nil);
    IEServerWindow := FindWindowEx(
      NextWin, 0, 'Internet Explorer_Server', nil
    );
  end;

  if Assigned(ActiveControl) then
    StatusBar1.SimpleText := 'ActiveControl = '
      + ActiveControl.Name
  else
    StatusBar1.SimpleText := 'ActiveControl = nil';
end;
```

Listing 6

The timer event is used to find the window since the handle may not be available straight away when the application is displayed.

We now need to handle mouse-down messages sent to the browser control window. Create an *OnMessage* event handler for the *TApplicationEvents* component:

```
procedure TDemoForm.ApplicationEvents1Message(var Msg: tagMSG;
  var Handled: Boolean);
begin
  if (Msg.hwnd = IEServerWindow) then
    if (Msg.message = WM_LBUTTONDOWN) or (Msg.message = WM_RBUTTONDOWN)
      or (Msg.message = WM_MBUTTONDOWN) then
      ActiveControl := WebBrowser1;
end;
```

Listing 7

No *TApplicationEvents*?

If you don't have *TApplicationEvents* with your version of Delphi simply create a private method with the same parameter list as *ApplicationEvents1Message* and implement it as in *listing 7*. Then create a *OnCreate* message handler for your form and assign your private method to the *Application.OnMessage* event within that message handler.

This code ensures the browser control is set as the form's active control whenever the control receives mouse-down events from the left, middle or right mouse buttons.

And that's it – clicking anywhere on the browser control activates it.

Gotcha #3

There has to be a problem doesn't there? This solution works fine providing there is only one web browser control on the form. If there is more than one then the solution fails. This is because the code that finds the browser control's window handle searches for a window class name – and the class name the same for each browser control instance. And that means only one browser control is found.

A fix?

A possible fix is to put each browser control in a separate *TPanel* and then find the required windows by using the appropriate panel's window handle in the *FindWindowEx* call that searches for the "Shell Embedding" window class (see *listing 6*). Of course, you'll need to track which window relates to which browser control.

Summary

In this article we noted that clicking in a *TWebBrowser* control does not make it the active control of the parent form. We developed some code to fix this problem by detecting a mouse click on the control's active document and setting the parent form's *ActiveControl* property to the browser control in response to the click. This was done by handling the browser object's *OnCommandStateChange* event and detecting a text selection. We observed that this solution is far from perfect because it doesn't work if:

- there is no document loaded in the browser;
- the user clicks in the border of an HTML document;
- the user clicks on a image or other non-text area in the display.

An alternative method that does not have the above problems was then presented. This method depended on finding the browser control's window handle and detecting mouse messages directed to that window. It works when no document is loaded and detects clicks anywhere in the control, including the scroll bars. However, it was noted that the solution failes if there are two browser controls on the form.

Demonstration code

Demo code to accompany this article is available for download.

The demo includes the complete source code of three programs: one for each of the solutions presented above along with one that illustrates the problem.

The code was developed using Delphi 7 Professional. It may need some minor changes to compile with earlier versions, but should compile on later Win32 personalities of the compiler without change, although this has not been tested.

This source code is merely a proof of concept and is intended only to illustrate this article. It is not designed for use in its current form in finished applications. The code is provided on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. The source code is released under the same *Creative Commons License* as this article. If you agree to all this then please download the code using the following link.

Download the demo code