How to run a single instance of an application (part 2 of 4)

An initial approach

An Initial Approach

Finding if your program is running

We use the *FindWindow* Windows API routine to search for a top level window of the required class. This routine can check for window captions, (unreliable because they can change while a program is running), the window class name (better) or both. We opt to check for the window class name regardless of the caption.

There is a problem with this approach though – by default Delphi uses a form's class name as the name of the associated window class. For example, if a form's class is named *TForm1* then its window class name is also *TForm1*. Since it's possible to have two or more different Delphi applications running at the same time, both of which use this window class name we need to give our program's main window a class name that is extremely unlikely to be used by another application. We do this by overriding the main form's *CreateParams* method as follows:

```
procedure TForm1.CreateParams(var Params: TCreateParams);
begin
  inherited;
  StrCopy(Params.WinClassName, cWindowClassName);
end;
```

Listing 2

cWindowClassName is a constant set to some suitable class name. To ensure uniqueness we could use a GUID such as `D9CE4126-B845-42B2-ABAE-3678FF6EC836`. However I find that a name based on company name, the name of the application and possibly the major version number is human friendly and is sufficiently unlikely to be used by other applications. An example is `DelphiDabbler.MyApp.2`.

Now that we have a suitable window class name we can find the main window handle of any already existing instance of our application using the following function:

```
function FindDuplicateMainWdw: HWND;
begin
  Result := FindWindow(cWindowClassName, nil);
end;
```

Listing 3

The function will return the handle of any top level window with the required class name. It will return 0 if there is no such window. So, if the function returns 0 we have no existing instance and can run our application, but if the function returns a non zero value an existing instance is already running and we should terminate our current instance.

We need to be **very careful** where we place this test – if we run the check after we have created our application's main window the function will always return a window handle, and this may be the handle of our own window! Therefore it is important to run this check before the application creates its main form. We can do this by performing the check in the project (`.dpr`) file, before the main form is created. The project file can be modified as follows:

```
function CanStart: Boolean;
var
  Wdw: HWND;
begin
  Wdw := FindDuplicateMainWdw;
```

```
  if Wdw = 0 then
    // no instance running: we can start our app
    Result := True
  else
    // instance running: try to pass command line to it
    // terminate this instance if this succeeds
    // (SwitchToPrevInst routine explained later)
    Result := not SwitchToPrevInst(Wdw);
end;

begin
  if CanStart then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end;
end.
```

Listing 4

CanStart returns true if there is no other instance of the application (i.e. no top level window has been found with the main window class name). The application instantly terminates without ever displaying a window if *CanStart* returns false.

If a previous instance has been found we attempt to switch to it using the *SwitchToPrevInst* function (see *Listing 11*). We will discuss this function later in the article. For now it suffices to know that the function activates the main window of the previous application instance and attempts to pass any command line parameters to it. The function returns true if this attempt succeeds and false if it fails. Note that we go ahead and start our instance of the application if the attempt to pass the command line to the previous instance fails. This is a secure way of ensuring our command line gets processed.

That just about wraps up how to start a single instance of the application. Now we need find out how to pass any command line data to a previous instance.

Passing the command line to a previous instance

When the program detects that another instance is running it must pass its command line parameters to that program before terminating itself. This is taken care of in the *SendParamsToPrevInst* routine which sends the parameters to the main window of the existing application instance by using the *WM_COPYDATA* message.

Passing data to another application with WM_COPYDATA

This message is provided by Windows for the purpose of passing data across process boundaries – i.e. for sending data to other applications. Because the message sends data between address spaces we have to be careful to adhere the rules governing its use. The rules are:

- We can't pass pointers, only actual data.
- The sending application must allocate and free the memory required for the data.
- The receiving application must not free the data.
- The receiving application must copy the data to local storage while handling the message if the data is to be accessed after the message handler has returned.
- We must use *SendMessage* and not *PostMessage* to send the message.

The *WM_COPYDATA* message uses a structure of type *TCopyDataStruct* to store and describe the data to be sent. This structure has three fields:

- *dwData: DWORD* – holds a user defined 32 bit value.
- *lpData: Pointer* – points to a user defined data block (used if more than 32 bits of data are to be sent).
- *cbData: DWORD* – is the size of the user defined data block in bytes (zero if the data block is not used).

SendParamsToPrevInst function

As already noted the *SendParamsToPrevInst* function is used when we have to pass command line parameters to a previous application instance. The function returns true if the receiving application processes

the *WM_COPYDATA* message properly and false if the message is not handled or if the receiving application reports an error. *Listing 5* gives the routine's implementation.

```
function SendParamsToPrevInst(Wdw: HWND): Boolean;
var
  CopyData: TCopyDataStruct;
  I: Integer;
  CharCount: Integer;
  Data: PChar;
  PData: PChar;
begin
  CharCount := 0;
  for I := 1 to ParamCount do
    Inc(CharCount, Length(ParamStr(I)) + 1);
  Inc(CharCount);
  Data := StrAlloc(CharCount);
  try
    PData := Data;
    for I := 1 to ParamCount do
    begin
      StrPCopy(PData, ParamStr(I));
      Inc(PData, Length(ParamStr(I)) + 1);
    end;
    PData^ := #0;
    CopyData.lpData := Data;
    CopyData.cbData := CharCount * SizeOf(Char);
    CopyData.dwData := cCopyDataWaterMark;
    Result := SendMessage(
      Wdw, WM_COPYDATA, 0, LPARAM(@CopyData)
    ) = 1;
  finally
    StrDispose(Data);
  end;
end;
```

Listing 5

The function first bundles the command line parameters into a single data structure. The data structure we have chosen is a #0 separated sequence of the program's parameter strings, terminated by a further #0 character. (This idiom is commonly used in Windows to record a list of string values). For example the three parameters

`'One'`, `'Two'` and `'Three'`

would be stored as

`One#0Two#0Three#0#0`

Having allocated the required memory and stored the data we set *TCopyDataStruct.lpData* to point to it before recording the size of the data structure in *TCopyDataStruct.cbData*.

Unicode strings

SendParamsToPrevInst works correctly with Unicode strings.

Note that the *CharCount* local variable counts *characters*, not *bytes*. The two are not equivalent in Unicode strings.

The character count is what we need when we call *StrAlloc* to allocate storage for the parameter strings because it accepts the length of the required string in characters as its parameter.

However, the *cbData* field of *TCopyDataStruct* requires the size of the data in *bytes*, so we multiply *CharCount* by *SizeOf(Char)* to get the required size.

Note: When Unicode strings are being used, references to the `#0` character should be read as the *WideChar* zero character.

We use the *TCopyDataStruct.dwData* field to store a 32 bit *"watermark"* value that we use to validate the *WM_COPYDATA* message. While the use of such a watermark is not essential, it does provide a reality check on the data and indicates to the receiving application that the message and its data are legitimate. The watermark can be any 32 bit value – we use the value stored in the *cCopyDataWaterMark* constant.

SendMessage is now used to send the message to the existing application's main window. The address of the *TCopyData* structure is passed as the message's *LParam*. We set the function result to *True* if the message returns 1 and *False* otherwise.

Finally, after *SendMessage* has returned, we free the memory used to store the command line parameters.

The receiving application needs to know how to extract the data from the *WM_COPYDATA* message. We discuss how to do this next.

Handling WM_COPYDATA

As well as knowing how to send data to other instances of itself, our application also needs to know how to handle the *WM_COPYDATA* messsage and to decode the data. We do this in the program's main form by writing a standard *Delphi* message handler, which is declaraed in the form's class declaration as per *Listing 6*. *Listing 7* shows the implementation of the message handler.

```
procedure WMCopyData(var Msg: TWMCopyData);
  message WM_COPYDATA;
```

Listing 6

```
procedure TForm1.WMCopyData(var Msg: TWMCopyData);
var
  PData: PChar;  // walks thru data
  Param: string; // a parameter
begin
  // check watermark is valid
  if Msg.CopyDataStruct.dwData <> cCopyDataWaterMark then
    raise Exception.Create(
      'Invalid data structure passed in WM_COPYDATA'
    );
  // extract strings from data
  PData := Msg.CopyDataStruct.lpData;
  while PData^ <> #0 do
  begin
    Param := PData;
    // process the parameter:
    // (write this method to do required processing)
    ProcessParam(Param);
    Inc(PData, Length(Param) + 1);
  end;
  // set return value to indicate we handled message
  Msg.Result := 1;
end;
```

Listing 7

This message handler processes the data referenced by the *TCopyDataStruct* data structure. The *WM_COPYDATA* message has a pointer to the record in its *LParam* field. Rather than accessing the *LParam* value directly and type casting we use *TWMCopyData* to "crack" the message record – its *CopyDataStruct* field provides easy access to the data structure.

We first check the watermark stored in the structure's *dwData* member, raising an exception if it is not valid. Then we get a pointer the to #0 delimited list of parameters and walk the buffer, making a copy of each parameter before passing it to the *ProcessParam* method. The loop ends when a terminating #0 character is found. Before returning we set the message result to 1 to indicate it has been handled successfully.

ProcessParam

ProcessParam is just a placeholder for whatever processing is to be applied to each parameter.

Activating the previous instance

At the time a previous instance of an application is activated by our new instance there is no way of telling whether its main window is displayed normally, maximized, minimized or invisible. The window may be partially or fully hidden behind other windows. If it is minimized, invisible or fully or partially hidden there will be no visual feedback to the user of the application they believe they started. There are various different design decisions that could be taken to handle these situations. We will opt to ensure the window it is made prominent when it is activated. If the window is shown normally or is maximised we simply bring it to the front of the z-order. If, however, the window is minimized we restore it and if it is hidden we show it.

When an application starts and finds there is a previous instance, the previous instance's main window is sent a custom message telling it to "wake up". This message is defined in *Listing 8*:

```
const
  UM_ENSURERESTORED = WM_USER + 1;
```

Listing 8

We send this message in the *SwitchToPrevInst* routine mentioned earlier (see *Listing 11*).

The receiving application must handle and act on this message. We do this in the application's main form, again using a *Delphi* message handler. The handler's declaration is shown in *Listing 9* and it's implementation is in *Listing 10*.

```
procedure UMEnsureRestored(var Msg: TMessage);
  message UM_ENSURERESTORED;
```

Listing 9

```
procedure TForm1.UMEnsureRestored(var Msg: TMessage);
begin
  if IsIconic(Application.Handle) then
    Application.Restore;
  if not Visible then
    Visible := True;
  Application.BringToFront;
  SetForegroundWindow(Self.Handle);
end;
```

Listing 10

Using *Application.MainFormOnTaskbar*

If you have one of the later version of Delphi and have set *Application.MainFormOnTaskbar* to *True* then change

```
IsIconic(Application.Handle)
```

to

```
IsIconic(Application.MainForm.Handle)
```

in *Listing 10*.

Thanks to Ron Tadmor for pointing this out.

In the message handler we check to see if the application is minimized and restore it if so. We then ensure that the main form is visible. Next we bring the main form to the front of the z-order before calling the *SetForegroundWindow* API function to bring our window to the fore.

SetForegroundWindow Issues

Microsoft's documentation of this function indicates that it may not always actually bring the Window to the fore – the circumstances when it may not do this are quite complex. Please check the function's *documentation on MSDN* for further details.

Finishing touches

It only remains to implement the *SwitchToPrevInst* function we referred to earlier in the article. This function has two main purposes:

1. To ensure that any command line parameters are sent to the previous instance.
2. To activate the previous instance, ensuring its main window is prominently displayed.

The handle of the main window of the previous application instance is passed to the function as a parameter. This handle must represent an actual window and so must not be zero. Recall that we must return *True* if parameters are successfully sent to the previous instance and *False* on error. *Listing 11* has the implementation of *SwitchToPrevInst*:

```
function SwitchToPrevInst(Wdw: HWND): Boolean;
begin
  Assert(Wdw <> 0);
  if ParamCount > 0 then
    Result := SendParamsToPrevInst(Wdw)
  else
    Result := True;
  if Result then
    SendMessage(Wdw, UM_ENSURERESTORED, 0, 0);
end;
```

Listing 11

The first thing to note is that this function returns *True* only if command line parameters were sent to the previous instance successfully or if there are no parameters to send. We use *SendParamsToPrevInst* to send any parameters to the previous instance. That routine also returns *True* on success so we use its return value as our result. If we have successfully transferred any command line parameters (or there are none) we must

activate the previous instance. We do this by sending it a *UM_ENSURERESTORED* message. As we have seen in *Listing 10* the previous instance must respond to this message by ensuring its main window is restored and active.

Putting it all together

In this section we present the code of a skeleton *Delphi* project that draws together all the code discussed in this article. The code can be used as a template for a real application. The project comprises three modules:

1. `TemplateDemo.dpr` (*Listing 12*)
   A project file incorporating the required modifications to the Delphi-generated code.
2. `FmTemplate.pas` (*Listing 13*)
   A skeleton form unit containing the various message handlers and other necessary form code. The form itself contains no components.
3. `UStartup.pas` (*Listing 14*)
   A unit containing the various functions used to manage and activate the single application instance.

These modules are all included in the *demo code* that accompanies this article.

```
program TemplateDemo;

uses
  Forms,
  Windows,
  FmTemplate in 'FmTemplate.pas' {Form1},
  UStartup in 'UStartup.pas';

{$R *.res}

function CanStart: Boolean;
var
  Wdw: HWND;
begin
  Wdw := FindDuplicateMainWdw;
  if Wdw = 0 then
    Result := True
  else
    Result := not SwitchToPrevInst(Wdw);
end;

begin
  if CanStart then
  begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
  end;
end.
```

Listing 12

```
unit FmTemplate;

interface

uses
  Windows, Messages, SysUtils, Controls, Forms,
  UStartup;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    procedure ProcessParam(const Param: string);
    procedure UMEnsureRestored(var Msg: TMessage);
      message UM_ENSURERESTORED;
    procedure WMCopyData(var Msg: TWMCopyData);
        message WM_COPYDATA;
  protected
```

```
    procedure CreateParams(var Params: TCreateParams);
      override;
  public
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

{ TForm1 }

procedure TForm1.CreateParams(var Params: TCreateParams);
begin
  inherited;
  StrCopy(Params.WinClassName, cWindowClassName);
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  I: Integer;
begin
  for I := 1 to ParamCount do
    ProcessParam(ParamStr(I));
end;

procedure TForm1.ProcessParam(const Param: string);
begin
  { TODO : Code to process a parameter }
end;

procedure TForm1.UMEnsureRestored(var Msg: TMessage);
begin
  if IsIconic(Application.Handle) then
    Application.Restore;
  if not Visible then
    Visible := True;
  Application.BringToFront;
  SetForegroundWindow(Self.Handle);
end;

procedure TForm1.WMCopyData(var Msg: TWMCopyData);
var
  PData: PChar;
  Param: string;
begin
  if Msg.CopyDataStruct.dwData <> cCopyDataWaterMark then
    raise Exception.Create(
      'Invalid data structure passed in WM_COPYDATA'
    );
  PData := Msg.CopyDataStruct.lpData;
  while PData^ <> #0 do
  begin
    Param := PData;
    ProcessParam(Param);
    Inc(PData, Length(Param) + 1);
  end;
  Msg.Result := 1;
end;

end.
```

Listing 13

```
unit UStartup;

interface

uses
  Windows, Messages;

const
```

```
  // Name of main window class
  cWindowClassName = 'DelphiDabbler.SingleApp.Demo';
  // Any 32 bit number here to perform check on copied data
  cCopyDataWaterMark = $DE1F1DAB;
  // User window message handled by main form ensures that
  // app not minimized or hidden and is in foreground
  UM_ENSURERESTORED = WM_USER + 1;

function FindDuplicateMainWdw: HWND;
function SwitchToPrevInst(Wdw: HWND): Boolean;

implementation

uses
  SysUtils;

function SendParamsToPrevInst(Wdw: HWND): Boolean;
var
  CopyData: TCopyDataStruct;
  I: Integer;
  CharCount: Integer;
  Data: PChar;
  PData: PChar;
begin
  CharCount := 0;
  for I := 1 to ParamCount do
    Inc(CharCount, Length(ParamStr(I)) + 1);
  Inc(CharCount);
  Data := StrAlloc(CharCount);
  try
    PData := Data;
    for I := 1 to ParamCount do
    begin
      StrPCopy(PData, ParamStr(I));
      Inc(PData, Length(ParamStr(I)) + 1);
    end;
    PData^ := #0;
    CopyData.lpData := Data;
    CopyData.cbData := CharCount * SizeOf(Char);
    CopyData.dwData := cCopyDataWaterMark;
    Result := SendMessage(
      Wdw, WM_COPYDATA, 0, LPARAM(@CopyData)
    ) = 1;
  finally
    StrDispose(Data);
  end;
end;

function FindDuplicateMainWdw: HWND;
begin
  Result := FindWindow(cWindowClassName, nil);
end;

function SwitchToPrevInst(Wdw: HWND): Boolean;
begin
  Assert(Wdw <> 0);
  if ParamCount > 0 then
    Result := SendParamsToPrevInst(Wdw)
  else
    Result := True;
  if Result then
    SendMessage(Wdw, UM_ENSURERESTORED, 0, 0);
end;

end.
```

Listing 14

The above approach to the problem works and provides a template for use in any application. However the source code needs to be ammended for each application. This is crying out for a reusable, preferably object oriented, solution – but can we find one?

Well partly! In the *next part* of the article we develop at an object oriented solution that, while still requiring some modification of the main form and project file, does at least move most of the code to an extensible

class.