How to catch files dragged and dropped on an application from Explorer

Contents

Why do it?

Many programs allow the user to open files by dragging them from Explorer and dropping them on the program's window. This is often more convenient to the user than using the normal *File* | *Open* menu option or toolbar button since it misses out the step of navigating an *Open File* dialog box. Adding similar support to your program makes it appear more professional.

How it's done

There are two main ways to add support to a Windows application. The first is to use Windows drag and drop API functions and handle a related Windows message. The second method uses OLE (COM), and additionally supports inter and intra-application drag and drop. We'll explore the first, and most simple, of these methods here.

OLE Drag & Drop

OLE drag and drop is explored in the article "*How to receive data dragged from other applications*".

Overview

File drag and drop is not enabled in Windows applications by default. To support it we need to tell Windows we want to be notified about file drops, and nominate a window to receive the notifications. The nominated window has to handle the *WM_DROPFILES* message that Windows sends us when a drop occurs. The message supplies us with a "drop handle" that we pass to various API routines to get information about what was dropped. When we've finished good manners require that we tell Windows we no longer need to be told about drops.

Windows Vista Security Problems

For security reasons Windows Vista may disallow drag and drop between windows that have different security permissions. As a consequence *WM_DROPFILES* messages may be blocked.

You can read more about the problem on this *Microsoft forum*

Here is a step by step guide to what we need to do in a Delphi application to handle dropped files:

1. Use the `ShellAPI` unit to get access to the required API routines:

```
uses ShellAPI
```

Listing 1

2. Call *DragAcceptFiles*, passing the handle of the window that is to receive *WM_DROPFILES* messages, along with a flag to indicate we want to receive notifications. For example for our main program form to receive the messages:

```
DragAcceptFiles(Form1.Handle, True);
```

Listing 2

3. Handle the *WM_DROPFILES* message. In Delphi we can declare a message handler for this event in the our main form's class declaration:

```
procedure WMDropFiles(var Msg: TWMDropFiles); message WM_DROPFILES;
```

Listing 3

4. In the *WM_DROPFILES* message handler use the *DragQueryXXX* API functions to retrieve information about the file drop – see the explanation *below*.

5. Notify Windows we have finished with the supplied drop handle. We do this by calling *DragFinish*, passing it the drop handle we retrieved from the *WM_DROPFILES* message. This frees the resources used to store information about the drop:

```
DragFinish(DropH);
```

Listing 4

6. When we are closing down our application we call *DragAcceptFiles* again, but this time with a final parameter of False to let Windows know we're no longer interested in handling file drops:

```
DragAcceptFiles(Form1.Handle, False);
```

Listing 5

Getting information about dropped files

We use two API function to get information about dropped files – *DragQueryFile* and *DragQueryPoint*. Both of these require the drop handle from the *WM_DROPFILE* messages.

DragQueryFile – jack of all trades

Like many Windows API functions, *DragQueryFile* can perform several functions depending on the parameters passed to it. This makes it hard to remember exactly how to use it. Its parameters (in Delphi-speak) are:

- *DropHandle: HDROP* – the drop handle provided by the *WM_DROPFILES* message.
- *Index: Integer* – the index of the file to query in the list of dropped files.
- *FileName: PChar* – pointer to a dropped file name.
- *BufSize: Integer* – size of buffer for the above.

The three functions the function performs are:

1. *Finding the number of files dropped.*
   We get this information by passing $FFFFFFFF as the *Index* parameter, **nil** as the file name parameter and 0 as the *BufSize* parameter. The return value is the number of files dropped. Obvious isn't it!

2. *Finding the size of buffer needed to store the file name at a specific (zero based) index.*
   We pass the file's index number in the *Index* parameter, **nil** for the file name and 0 for the buffer size. The function then returns the buffer size required.

3. *Fetching the name of the file at a given index.*
   We pass the index number, a *PChar* buffer large enough to hold the file name + the #0 terminator, and the actual buffer size.

All this makes for extremely unreadable code which, later in the article, we'll hide away in a class.

DragQueryPoint

This function is quite an anti-climax after *DragQueryFile* – it simply does what it says and provides the mouse cursor position where the files were dropped. The parameters are:

- *DropHandle: HDROP* – the drop handle from *WM_DROPFILES*.
- ***var** Point: TPoint* – reference to a *TPoint* structure that receives the required point. The function returns non-zero if the drop was in the window's client area and zero if it wasn't.

Putting it all together

Let's pull all we've discovered together in a skeleton Delphi application whose main form, Form1, needs to be able to catch and process dropped files.

In our form creation event handler we register our interest with Windows:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  // ... other code here
  DragAcceptFiles(Self.Handle, True);
  // ... other code here
end;
```

Listing 6

The meat of the processing comes in the *WM_DROPFILES* event handler:

```
procedure TForm1.WMDropFiles(var Msg: TWMDropFiles);
var
  DropH: HDROP;               // drop handle
  DroppedFileCount: Integer;  // number of files dropped
  FileNameLength: Integer;    // length of a dropped file name
  FileName: string;           // a dropped file name
  I: Integer;                 // loops thru all dropped files
  DropPoint: TPoint;          // point where files dropped
begin
  inherited;
  // Store drop handle from the message
  DropH := Msg.Drop;
  try
    // Get count of files dropped
    DroppedFileCount := DragQueryFile(DropH, $FFFFFFFF, nil, 0);
    // Get name of each file dropped and process it
    for I := 0 to Pred(DroppedFileCount) do
    begin
      // get length of file name
      FileNameLength := DragQueryFile(DropH, I, nil, 0);
      // create string large enough to store file
      // (Delphi allows for #0 terminating character automatically)
      SetLength(FileName, FileNameLength);
      // get the file name
      DragQueryFile(DropH, I, PChar(FileName), FileNameLength + 1);
      // process file name (application specific)
      // ... processing code here
    end;
    // Optional: Get point at which files were dropped
    DragQueryPoint(DropH, DropPoint);
    // ... do something with drop point here
  finally
    // Tidy up - release the drop handle
    // don't use DropH again after this
    DragFinish(DropH);
  end;
  // Note we handled message
  Msg.Result := 0;
end;
```

Listing 7

First we record the drop handle in a local variable for convenience and then make the first of those confusing calls to *DragQueryFile* to get the number of files dropped. Now we loop through each file by index (zero based, remember) and get the file names. For each file name we first get the required

Using SetLength

SetLength(Str, N) creates a buffer large enough for *N* characters plus the terminating #0.

buffer size using the second form of *DragQueryFile*, then create a sufficiently large string. We finally read the string into the buffer using the third form of *DragQueryFile*.

After reading all the file names we then get the drop point. Once all the processing is done we call *DragFinish* to release the drop handle. Notice we do this in a **finally** section since the drop handle is a resource that we need to ensure is freed, even if there is an exception. We end by settting the message result to 0 to indicate to Windows that we have handled it.

Finally, we unregister our interest in file drops in the form destruction event handler:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  // ... other code here
  DragAcceptFiles(Self.Handle, False);
  // ... other code here
end;
```

Listing 8

The Delphi Way

If you agree with me that all the messing about with API routines rather spoils our Delphi code, what better than to make a helper class to hide a lot of the mess. Here's a small class that can be created and destroyed within the *WM_DROPFILES* message handler. The class hides away a lot of the code, although we must still handle *WM_DROPFILES* ourselves. We must also notify Windows of our intention to accept drag drop.

The class's declaration appears in *Listing 9* while *Listing 10* has the implementation.

```
type
  TFileCatcher = class(TObject)
  private
    fDropHandle: HDROP;
    function GetFile(Idx: Integer): string;
    function GetFileCount: Integer;
    function GetPoint: TPoint;
  public
    constructor Create(DropHandle: HDROP);
    destructor Destroy; override;
    property FileCount: Integer read GetFileCount;
    property Files[Idx: Integer]: string read GetFile;
    property DropPoint: TPoint read GetPoint;
  end;
```

Listing 9

```
constructor TFileCatcher.Create(DropHandle: HDROP);
begin
  inherited Create;
  fDropHandle := DropHandle;
end;

destructor TFileCatcher.Destroy;
begin
  DragFinish(fDropHandle);
  inherited;
end;

function TFileCatcher.GetFile(Idx: Integer): string;
var
  FileNameLength: Integer;
begin
  FileNameLength := DragQueryFile(fDropHandle, Idx, nil, 0);
  SetLength(Result, FileNameLength);
  DragQueryFile(fDropHandle, Idx, PChar(Result), FileNameLength + 1);
end;

function TFileCatcher.GetFileCount: Integer;
begin
  Result := DragQueryFile(fDropHandle, $FFFFFFFF, nil, 0);
end;
```

```
function TFileCatcher.GetPoint: TPoint;
begin
  DragQueryPoint(fDropHandle, Result);
end;
```

Listing 10

There is not much to explain, given the code we have already seen. The constructor takes a drop handle and records it. The drop handle is "freed" in the destructor with a call to *DragFinish*. The list of dropped files, the number of files dropped and the drop point are all provided as properties. The assessor methods for the properties are simply wrappers round the *DragQueryFile* and *DragQueryPoint* API functions.

Let us rewrite the *WM_DROPFILES* message handler to use the new class:

```
procedure TForm1.WMDropFiles(var Msg: TWMDropFiles);
var
  I: Integer;                 // loops thru all dropped files
  DropPoint: TPoint;          // point where files dropped
  Catcher: TFileCatcher;      // file catcher class
begin
  inherited;
  Catcher := TFileCatcher.Create(Msg.Drop);
  try
    for I := 0 to Pred(Catcher.FileCount) do
    begin
      // ... code to process file here
    end;
    DropPoint := Catcher.DropPoint;
    // ... do something with drop point
  finally
    Catcher.Free;
  end;
  Msg.Result := 0;
end;
```

Listing 11

Going further

The *TFileCatcher* class could be extended to encapsulate all of the drop files functionality, hiding away all the API calls. To do this would require access to the form's window so we could intercept its messages and respond to *WM_DROPFILES*. One way to do this is to subclass the form's window. It is beyond the scope of this article to look at how that can be done. However, if you want to invetigate further, please check out my *Drop Files components*.

Demo Application

The source code of a demo application that uses the code reviewed here is available for download. The code has been tested with Delphi 4 and Delphi 7.

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 code may be used in any way providing it is not sold or passed off as the work of another person. If you agree to all this then please download the code using the following link.

Download the demo code