Drag & Drop Is Easy...

by Stuart Lunn

How to enable your application to become a client of Windows File Manager (and how to hook into the Windows Message Loop at the same time)

If an application has been made a client of Windows File Manager, you can drag a file from File Manager and drop it onto the application, just as you can move and copy files within File Manager by dragging them from one place to another. For example, if you drag a text file called TEMP.TXT from File Manager and drop it on the Delphi code editor the file will be opened with the name TEMP.TXT on the code page window tab (a very useful feature which I discovered by accident!). Incidentally, you will know if an application can accept dropped files because the cursor will change from the *No Stopping* sign to one of the document drop symbols.

While Delphi makes it easy to implement drag and drop between components, it does not automatically enable your application as a client of File Manager. But with a simple API call and capture of the Windows `WM_DROPFILES` message you can easily enable your application as a drag and drop client to File Manager. It's just a matter of knowing which units to add to the `Uses` clause, and what functions to call. All you need to do is:

- Add `SHELLAPI` to the `Uses` clause of the unit,
- Call the `DragAcceptFiles` API function to tell File Manager that your application will accept files dropped on it,
- Write code to process the dropped files.

In `ShellAPI` are held various function declarations for interfacing with File Manager. Somewhere early on you will need to call `DragAcceptFiles(handle, true)` with the parameters set as shown. The first parameter is the handle to the form and the second parameter is a flag to tell File Manager that the application will accept dropped files. If your application takes some time processing a dropped file, you may need to call this function with the flag set to false to stop files being dropped. When the processing has finished and you are ready to accept more files you can then set it to true.

Your application must also catch the Windows `WM_DROPFILES` message. A Windows message is a record structure that contains fields which hold information about the message. The fields most used are `wParam` (for "word parameter") which is 16 bits and `lParam` (for "long parameter") which is 32 bits.

If you declare the message with one of the newer record types defined by Microsoft, you can also get at the information in the message more easily by using fields with specific names and types. These specific record types are defined by Delphi with appropriate names. For example, when the `WM_SYSCOMMAND` message is received the `lParam` holds the coordinates of the cursor if the system menu was chosen using the mouse. Using the Delphi `TWMSysCommand` message the x and y coordinates can be referred to by the names `XPos` and `YPos`. One small note of caution: the Windows `WM_SYSCOMMAND` message names the `wParam` parameter as `wCmdType`, while Delphi names it `CmdType` (without the "w"). Refer to the *Component Writer's Help* for details of the messages.

To declare a message-handling method, you must declare a procedure in a protected part of the component's class definition. The convention for naming such a message handler is to give it a name after the message that it handles but without any underline characters. Following this convention we add a line similar to the following to the private section in the form's class declaration:

```
procedure wmDropFiles(var
  Message : TMessage);
  message WM_DROPFILES;
```

Now whenever the form receives a `WM_DROPFILES` message it will call its `wmDropFiles` method. The single parameter of type `TMessage` contains the details of the message. The code in the `wmDropFiles` method will need to handle the applicable fields in this structure.

Note that the `TMessage` type provides a general message structure. When you use the fields in this message structure you will probably need to cast them to the specific type that your application requires. Alternatively, if available you can use one of the special message types, as done for the `WM_SYSCOMMAND` message discussed below. Note that the declarations for the standard Windows messages are held in the `Messages` unit which is automatically added to the `Uses` clause of a new form.

The code in the `wmDropFiles` method must first find the number of files that have been dropped. To do this call the `DragQueryFile` API function as follows:

```
iNumberDropped :=
  DragQueryFile(
  THandle(message.wParam),
  $FFFF, NIL, 0);
```

The first parameter is the handle to the memory block that is set up by File Manager to contain the list of files (including paths) that were

selected by the user. This handle is held in the `wParam` field of the message and must be cast to type `THandle`.

The second parameter holds either the (zero based) file number to retrieve or the hex value $FFFF; if the latter is used the function will return the total number of files that were dropped and in this case the third parameter is NIL and the fourth parameter is 0.

Once you have the number of files that were dropped you can then process each file in turn. Use `DragQueryFile` again but this time to get a file name:

```
DragQueryFile(
  THandle(message.wParam), II,
  @szPathName, maxPathSize);
```

The first parameter is again the handle. The second parameter now indicates which file number to retrieve. The third parameter points to a buffer where the file name should be copied, and the fourth parameter gives the size of this buffer. Because the Windows API uses C-style strings, the file name and the buffer size must allow for the NULL terminating character. If needed you can also use `DragQueryFile` to supply the size of the path name (do a search for `DragQueryFile` in the Delphi on-line help). `DragQueryPoint` will give you the coordinates of the point at which the files were dropped, so you could perform different actions dependent upon where the user released the left mouse button.

Once you have the path name you can change it to a Pascal type string using the `StrPas` function:

```
szPathName :=
  strPas(@szPathName);
```

You can then process each file as required. If the processing takes time you may wish to temporarily stop the user dropping further files by issuing the call

```
DragAcceptFiles(handle, false).
```

Finally, to free the block of memory containing the selected file names call `DragFinish`:

```
DragFinish(
  THandle(message.wParam));
```

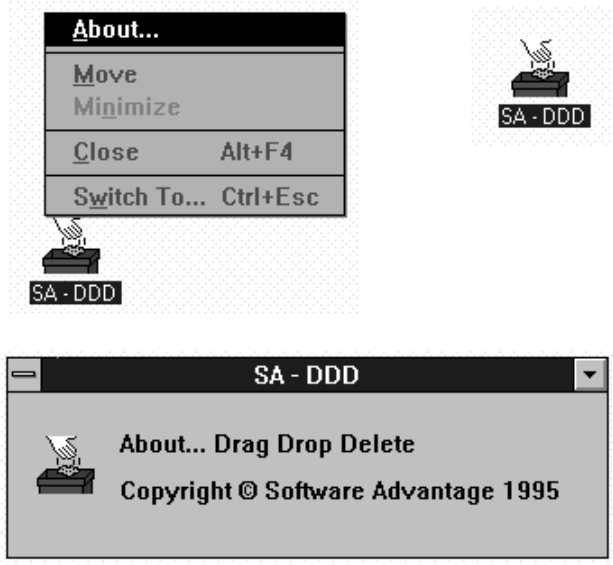
with the handle to the memory block as the only parameter. If you don't free the memory every drag operation will take a new slice of memory, and this will only be freed when File Manager is closed.

➤ *Figure 1*

Top right: the SA-DDD icon which remains on top of all other windows, waiting for files to be dropped onto it.

Top left: modified system menu.

Bottom: About box.



A Simple Example

Presented in Figure 1 is a noddy little application called SA-DDD (for Software Advantage Drag Drop Delete: if Microsoft and the other big boys push their names then why not the rest of us!). By the way, don't read the name of the application as sad, it's the basis for a useful application not a sad application!

All that SA-DDD does is display the name of any file that is dropped on it. A single line change will enable it to delete a file that is dropped on it. (As you can imagine, during testing the latter functionality is a real pain since File Manager has got to be kept topped up with garbage files for deleting – hence the trivial message box reporting the name of the dropped file. You can replace this with your own code.)

The application is started as an icon since no interactive functionality is required and it also takes less space on the screen.

The properties for the form are set as follows:

```
biMaximize     false
BorderStyle    bsSingle
WindowState    wsMinimized
```

You can also add your own icon and labels to the form. Use `Options|Project|Application` to set your run-time icon.

Referring to the code on the free disk with this issue, the first thing to notice is that the `ShellAPI` unit is added to the `Uses` clause: this unit contains the declarations for the functions needed to interact with File Manager. Next, in the private declarations section of the `TfrmDragPrint` class are declared two procedures:

```
procedure wmDropFiles(
  var Message : TMessage);
  message WM_DROPFILES;
procedure wmSysCommand(
  var Message : TWMSysCommand);
  message WM_SYSCOMMAND;
```

If you need to be able to derive a new class from `TfrmDragPrint` you may want to place these two declarations in a protected part of the class declaration.

The first method is used to process the `WM_DROPFILES` messages: the message is passed in the variable `Message` which is of the general

message type TMessage; thus the message fields may need to be cast to the appropriate type before use.

The second method is used to process the WM_SYSCOMMAND messages, but this time the message is of type TWMSysCommand so the names and types of the message fields should be more directly relevant.

The FormCreate method first checks that no more than one instance of the application is running. If hPrevInst is not zero then you know that there is already an instance of the application running, in which case you can terminate the current instance.

Next, the characteristics of the form are set with a call to the SetCharacteristics procedure. If you are like me then you will work most of the time with File Manager maximized. Consequently, when you click on a file in File Manager you will lose any other window that was visible. To get around this you can make the form stay on top by calling SetWindowPos with the second parameter set to HWND_TOPMOST. The form will then keep its topmost position even when another application has focus. (This can create a small problem if the form is not iconized when you drag a file onto it. The message box will be underneath the form and will be difficult for the user to get at! In this circumstance you could make your form invisible or make the message box stay on top.)

Rather than create a menu on the form I decided that the system menu would be sufficient with some small changes. Via a handle to the system menu the Size and Maximize items are removed and the wording of the Restore item is set to About; the API calls needed to do this are DeleteMenu, ModifyMenu and InsertMenu. This permits the form to be used as an About box – you could use it to set options if you prefer. If you need to restore the system menu to its original contents you can call GetSystemMenu with the second parameter set to true. If you want to add your own items to the system menu ensure that you use id numbers for your commands with values less than $F000; if you don't then your commands could conflict with Windows' own id numbers. The value of CmdType when the WM_SYSCOMMAND message is received tells you what menu item was selected.

If you need to make use of the default handling of messages then call Inherited. This will cause the default handling to take place by calling the original method (in this case the one with the same message index). I have used this simple technique in the wmSysCommand method (I noticed that under normal circumstances after minimising the About box the icon lost its topmost attribute and the system menu reverted to the original – clearly Delphi is up to no good here, or perhaps it's by design!).

In the wmSysCommand method the Message parameter is of type TWMSysCommand. Instead of using wParam and possibly casting to the type you require, you can refer to the fields directly. When the WM_SYSCOMMAND message is received with CmdType equal to SC_ICON the CloseWindow API function is used to iconize the application rather rely on the default. Be sure to use Inherited on commands that you are not processing otherwise you will disable all other commands on the system menu.

Finally, in the wmDropFiles method the processing of the dropped files takes place. Not a lot can be said here that isn't obvious: the method just loops over all the files that were dropped and presents their names in a message box. To do more, I suggest that you replace the message box with a call to a function that does all the work.

Other Ideas

By hooking into the message loop you can create some very useful things. For example, you may wish to:

- Develop a drag and delete application with animation. Hint: you will need a timer and several icons showing slightly different positions. To do this and make your application a source for drag and drop operations see: *Drop Everything: How to Make Your Application Accept and Source Drag-and-Drop Files* by J Richter, Microsoft Systems Journal, June 1992.
- Develop an application that will accept text files and show them in a read only mode without the cursor, and as the user presses the arrow keys the screen scrolls – a bit like reading a Windows help file. You could also implement screen scrolling using mouse movement perhaps with a mouse button or a key pressed at the same time.
- Develop an application that will permit a text file to be dropped on an icon and printed. This would be very useful for readme files – much simpler than opening Notepad or Write. This could be built with the text reader mentioned above. Default mode (read or print) could be stored in an INI file, the contents of which could be set using a dialog in place of or in addition to an about box.

If you have no need for the icon to be anything other than an icon you can trap the WM_QUERYOPEN message and override its normal handling. One technique is to have a method called QueryOpen that is called when your application receives the WM_QUERYOPEN message. This method need do no more than just exist: it doesn't need to do anything. By trapping the WM_QUERYOPEN message it will stop the default handler being used.

As a time-served VB developer, I was surprised to find this project so easy. To do this sort of thing with VB you would need a VBX kludge. Without a doubt, the designers of Delphi have given application developers an incredible degree of flexibility without needing to write clever code. I'm now off to write a Delphi application that will help me to win the British National Lottery...

Stuart Lunn can be contacted at 100031.215@compuserve.com by email. The complete source files for the application discussed in this article are of course on the free disk with this issue.