Starting with Drag & Drop in Lazarus and Delphi

Michaël Van Canneyt

January 16, 2016

Abstract

In this article we show how to implement drag and drop operations in your article.

1 Introduction

Drag and Drop is a common operation in a well-thought of GUI. It makes life for the user easier. It is therefor good to have some knowledge on how to implement drag and drop for simple (or less simple) cases. In this article, we'll look at 2 types of Drag&Drop:

1. Files dragged onto your application from a file manager program (such as file explorer on windows): The expectation of the user is then that the application will open the files if it supports the file format.
2. Drag and drop between visual elements (controls) in an application. For instance, the user drags selected text to another location; Or drags items from one list to another list.

Both kinds of drag&drop must be explicitly enabled and programmed, and they are handled in a completely different way.

Drag and drop is implemented in the same way in Lazarus and Delphi for the second case (drag and drop within an application), but for the first case, the mechanisms are different.

2 Drag and drop of files in Delphi

In Deplhi, Drag&Drop of files must be implemented manually.

In the `OnCreate` event (or `OnShow`), the `DragAcceptFiles` call from the `WinAPI.ShellAPI` unit must be called to notify Windows that the form accepts files. This call looks as follows:

```
procedure DragAcceptFiles(Wnd: HWND; Accept: BOOL); stdcall;
```

The following code will then enable dropping of files on a form:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  DragAcceptFiles(Handle,True);
end;
```

To actually accept the files, the `WM_DROPFILES` message that is sent to the application handle must be handled. There are 2 possible options. The first is to assign the `Application.OnMessage` handler in the `OnCreate` event handler of the form:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
  DragAcceptFiles(Handle,True);
  Application.OnMessage:=HandleDrops
end;

procedure TMainForm.HandleDrops(var Msg: tagMSG; var Handled: Boolean)

begin
end;
```

The second method is to drop a `TApplicationEvents` component on the form, and to assign its `OnMessage` event handler (It has the same signature as the above `HandleDrops` procedure).

In the `HandleDrops` message handler, the actual file drop handling must be done. The file drop message can be processed with the `DragQueryFile` call. This call looks as follows:

```
function DragQueryFile(Drop: HDROP;
                       FileIndex: UINT;
                       FileName: LPWSTR;
                       cb: UINT): UINT; stdcall;
```

It serves 2 purposes:

1. Retrieve the number of files that was dropped (FileIndex equals $FFFFFFFF).
2. Retrieve the filenames of each of the files: FileIndex (≥ 0) indicates the index of the file.

The name of the file will be placed in a buffer, whose size must be reported in `cb`

Once the files were accepted, windows must be notified that the drop is completed, this happens with the `DragFinish` routine:

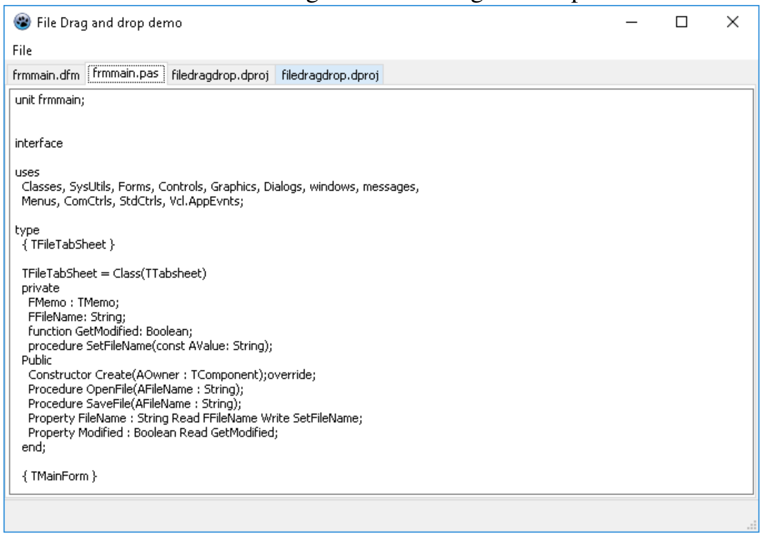
```
Procedure DragFinish(Drop: HDROP); stdcall;
```

Both calls accept as the first parameter a handle to a drag&drop structure. The Drag&Drop structure handle is passed in the windows message `WParam` field.

The following implementation will retrieve all files, and calls a `NewEditor` routine for each dropped file. This could be used in for example a small text editor application which - like so many editors - displays each file in a memo on a tab page:

```
procedure TMainForm.HandleDrops(var Msg: tagMSG; var Handled: Boolean);

Var
  NrFiles,I : Integer;
  Buffer : Array[0..255] of Char;


begin
  Handled:=Msg.Message=WM_DROPFILES;
  if not Handled then
    exit;
```

Figure 1: File drag and drop in action



```
unit frmmain;


interface

uses
  Classes, SysUtils, Forms, Controls, Graphics, Dialogs, windows, messages,
  Menus, ComCtrls, StdCtrls, Vcl.AppEvnts;

type
  { TFileTabSheet }

  TFileTabSheet = Class(TTabsheet)
  private
    FMemo : TMemo;
    FFileName: String;
    function GetModified: Boolean;
    procedure SetFileName(const AValue: String);
  Public
    Constructor Create(AOwner : TComponent);override;
    Procedure OpenFile(AFileName : String);
    Procedure SaveFile(AFileName : String);
    Property FileName : String Read FFileName Write SetFileName;
    Property Modified : Boolean Read GetModified;
  end;

  { TMainForm }
```

```
  NrFiles:=DragQueryFile(Msg.Wparam,$FFFFFFFF,Buffer,256);
  For I:=0 to NrFiles-1 do
    begin
    DragQueryFile(Msg.Wparam,I,Buffer,256);
    NewEditor(StrPas(Buffer));
    end;
  DragFinish (Msg.WParam);
end;
```

The end result of this code can be viewed in figure 1 on page 3.

3 Drag and drop of files in Lazarus

Handling drag and drop of files in Lazarus is quite easy. One property, and one event handler of TForm are involved:

AllowDropFiles when this property is set to `True`, it enables files to be dropped on the form. (When dragging files, the cursor will change shape to indicate this).

OnDropFiles This event is invoked when files have been dropped on the form. The event will not be invoked when 'AllowDropFiles' is set to `False`

When created by the IDE, the event handler looks as follows:

```
procedure TMainForm.FormDropFiles(Sender: TObject;
                                  const FileNames: array of String);

begin
end;
```

The `FileNames` parameter is an array of strings, which contains the full pathnames of all the files that were dropped on the form.

What is to be done with the dropped filenames depends on the application. Typically, it will open the file and display the contents.

If we repeat the idea of a small text editor application which - like so many editors - displays each file in a memo on a tab page, then the following code will open the file and edit it:

```
procedure TMainForm.FormDropFiles(Sender: TObject;
                                  const FileNames: array of String);

Var
  I : Integer;
begin
  // High(FileNames) is the last element in an open array
  For I:=0 to High(FileNames) do
    begin
    NewEditor(FileNames[i]);
    end;
end;
```

In the above code, the `FileNames` array is traversed, and for each file, a new editor is opened. That is all there is to drag&drop of files. In a real-world application, probably the extension of the file would be examined to see if it is a supported file format.

4 Simple drag and drop within an application

To dupport drag and drop within an application requires somewhat more properties than file drag and drop. These properties must be set on all the controls that are involved in the drag an drop operation(s).

DragMode This controls how a drag&drop operation is started: When set to `dmAutomatic`, then a drag&drop is started as soon as the user drags the mouse over the control. If it is set to `dmManual`, then a drag&drop operation must be started with the `StartDrag` method. This can be needed for instance to mark the difference between a drag&drop and a docking operation.

OnStartDrag This event is called whenever a drag&ldrop operation is started. This will change the cursor to a drag&drop cursor and initiates the other drag&drop events.

OnDragOver This event called when something is dragged over the control. It can be used to indicate whether the dragged item can actually be dropped on this control.

OnDragDrop This event is called when something is dropped on the control.

DragCursor Cursor to display during the drag&drop operation.

DragKind This can be used to decide whether dragging the mouse starts a drag&drop operation or a docking operation.

Typical use for these events is to implement a drag&drop operation between 2 lists (be it a listview, listbox or tree), or within a single list, to reorder the items in a list. This will be demonstrated with a small application using 2 listboxes. Moving the items between the listboxes can be done using buttons, but also with drag and drop. The items can also be re-ordered in a listbox, again using drag-and-drop.

The demo application has 2 listboxes (called `LBLeft` and `LBRight`. Both listboxes must have their `DragMode` property set to `dmAutomatic`. To make it more interesting, they have their `MultiSelect` set to `True`.

For this simple example, the `OnStartDrag` event will not be used. But the `DragOver` event will be used. This event has the following parameters:

Sender Is the control that triggered the event (`LBLeft` in this case).

Source This is normally the control that initiated the Drag&Drop operation (but it can be manipulated).

X,Y The coordinates of the mouse relative to the control.

Accept A boolean parameter that must be set by the routine to indicate whether a drop on this control is allowed.

Knowing this, the `OnDragOver` event of the `LBLeft` listbox can be codes as follows to accept drops from itself, or from the right combobox:

```
procedure TMainForm.LBLeftDragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept:=(Source=LBLeft) or (Source=LBRight);
end;
```

The same code can be used for the `LBRight` listbox.

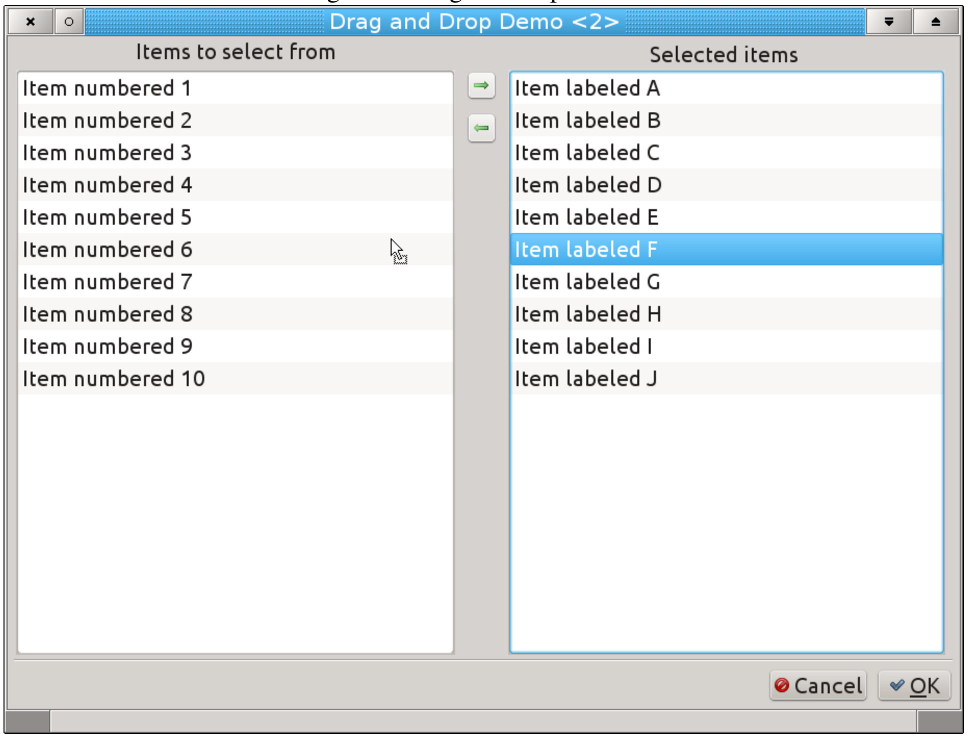
When the user relased the mouse at the end of a drag&drop operation, the `OnDragDrop` event is triggered. This event has amost the same signature as the DragOver event handler: it doesn't have the `Accept` parameter. For the left listbox, this means the eventhandler can be coded as follows:

```
procedure TMainForm.LBLeftDragDrop(Sender, Source: TObject; X, Y: Integer);

Var
  LBFrom,LBTO : TListBox;

begin
  LBFrom:=(Source as TListBox);
  LBTo:=(Sender as TListBox);
  If (LBTo=LBFrom) then
    MoveItemsToIndex(LBTo,LBTo.GetIndexAtXY(X,Y))
  else
    MoveItemsToListBox(LBTo,LBFrom,LBTo.GetIndexAtXY(X,Y));
end;
```

The `MoveItemsToIndex` and `MoveItemsToListbox` handle the reordering of the items and copying of items, respectively. The `X` and `Y` parameters are used to determine at what position the items should be inserted.

That is all there is to it. The Lazarus application in action can be seen in figure 2 on page 6.

Figure 2: Drag and drop in action



Item numbered 1
Item numbered 2
Item numbered 3
Item numbered 4
Item numbered 5
Item numbered 6
Item numbered 7
Item numbered 8
Item numbered 9
Item numbered 10

Item labeled A
Item labeled B
Item labeled C
Item labeled D
Item labeled E
Item labeled F
Item labeled G
Item labeled H
Item labeled I
Item labeled J

5 More advanced drag and drop

As shown, drag and drop can be implemented rather easily for simple cases: the source and target control are on the same form, there are only 2 controls, so checking is easy. But when multiple forms are in play, or there are many controls, some of them created dynamically, then the implementation becomes more difficult.

Drag&Drop operations are handled internally in the LCL and VCL using a `TDragObject` class. The LCL and VCL differ somewhat in the descendent objects that they make available, but other than that the mechanisms are identical.

The VCL and LCL allow the programmer to create a custom instance of `TDragObject` when the drag operation is started: the `OnStartDrag` event handler can be used for this. This instance is then used as the `Source` parameter of the `OnDragOver` and `OnDragDrop` events, instead of the control that started the drag operation.

By creating a descendent of the `TDragObject` class, it is possible to pass any desired information to the event handlers used in the Drag&Drop operation.

To demonstrate this, we will extend the listbox example: First of all we'll make it an application with multiple forms:

- A form with a listbox (`TListBoxForm`).
- A form with a memo (`TMemoForm`).
- A form with a listview (`TListViewForm`).
- A main form with a listbox, and 3 buttons to create the above secondary forms, as often as the user clicks on the button.

Each of the forms, when created, first sets the caption of the form to some unique text so the window can be identified by the user. Then it populates the control it has, with some items. Note that these secondary forms all contain controls that somehow can represent a list of strings, which the user can select. From each of these secondary forms, the user will be able to drag items from the list to the listbox on the main form.

In the main form, the procedures to create a secondary form are quite simple, and look like this:

```
procedure TMainForm.BListBoxWindowClick(Sender: TObject);
begin
  With TListBoxForm.Create(Self) do
    Show;
end;
```

Note that no reference to the created forms is kept: this means that the main form cannot directly check where a drag operation was initiated, and therefor cannot decide whether a drop on the listbox is allowed or not.

How can the Memo, Listview or ListBox controls on the secondary forms communicate to the main form what items are dragged by the user ?

For this, a descendent of `TDragObject` is created which can contain the items that will be dragged. Since all controls show lists of strings, this object will have a property called `Items` of type `TStrings`:

```
// For Delphi, the parent class is TDragControlObject
TStringsDragObject = Class(TDragObjectEx)
private
  FItems: Tstrings; // to keep the items in.
  procedure SetItems(const AValue: Tstrings);
Public
  Constructor Create(AControl : TControl); override;
  Destructor Destroy; override;
  Property Items : TStrings Read FItems Write SetItems;
end;
```

Note that the base object for Delphi is named differently than in Lazarus `TDragControlObject` instead of `TDragControlEx`. The constructor with the `TControl` typed parameter is introduced in different classes. Other than that, the code is identical.

The methods of this class are self-explanatory, they just do some memory management.

```
procedure TStringsDragObject.SetItems(const AValue: Tstrings);
begin
  if FItems=AValue then exit;
  FItems.Assign(AValue);
end;

constructor TStringsDragObject.Create(AControl : TControl);
begin
  inherited Create(AControl);
  FItems:=TStringList.Create;
end;

destructor TStringsDragObject.Destroy;
```

```
begin
  FreeAndNil(FItems);
  inherited Destroy;
end;
```

This object is implemented in a separate unit: `dragdroplist`, which is used in all forms of the project.

An instance of this object can be created in the `OnStartDrag` event handlers of the listview, listbox and memo controls of the secondary forms. When the object is created, it is filled with the currently selected items in the control.

For the `ListBoxForm`, the `OnStartDrag` event handler that creates an instance of this class looks as follows:

```
procedure TListBoxForm.LBItemsStartDrag(Sender: TObject;
   var DragObject: TDragObject);

Var
  SDO : TStringsDragObject;
  I : Integer;

begin
  SDO:=TStringsDragObject.Create(LBItems);
  DragObject:=SDO;
  For I:=0 to LBItems.Count-1 do
    If LBItems.Selected[i] then
      SDO.Items.Add(LBItems.Items[i]);
end;
```

After creating an instance of the `TStringDragObject` class, its Items property is filled with the selected items from the listbox.

Similar code can be created for the listview and memo forms.

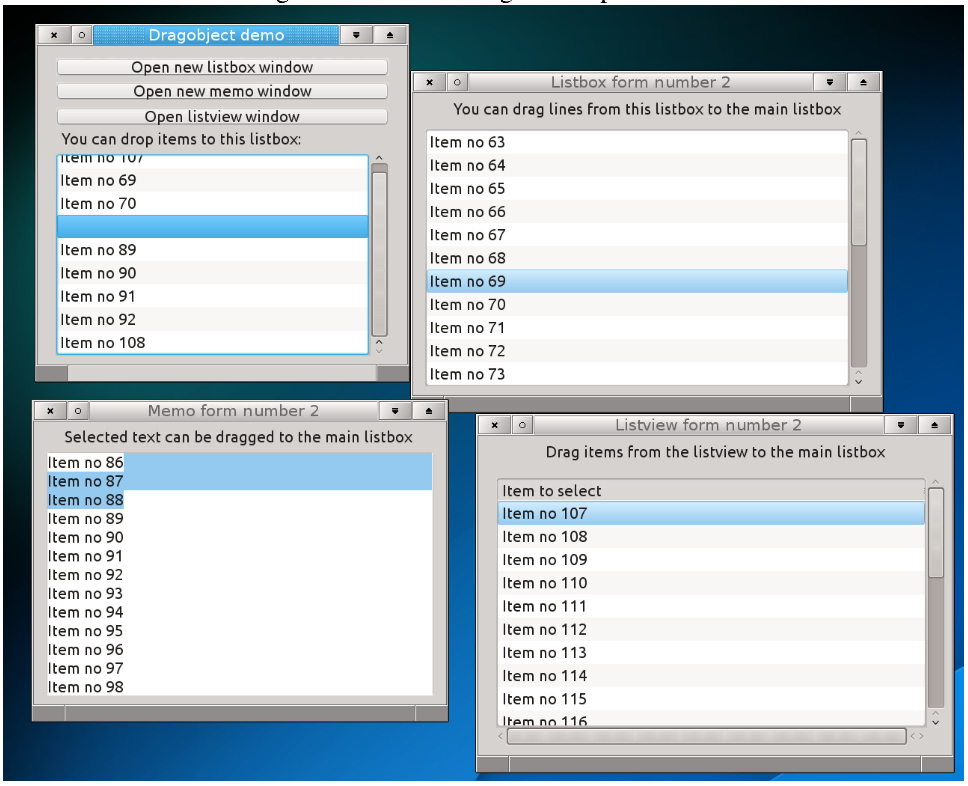
The start of the drag operation is herewith modified, and now the drop end of the operation must still be handled. For this, the `OnDragOver` event of the main listbox must be modified, so it checks that the `Source` parameter is a `TStringDragObject`, in which case it knows it can accept the items, and therefor a drop is allowed:

```
procedure TMainForm.LBMainDragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept:=Source is TStringsDragObject;
end;
```

All that remains to be done, is implementing the actual drop. Here again, the `Source` parameter will be the `TStringDragObject`, and this can be used to determine the items that must be added to the listbox:

```
procedure TMainForm.LBMainDragDrop(Sender, Source: TObject; X, Y: Integer);

Var
  I,L : Integer;
  SDO : TStringsDragObject;

begin
```

Figure 3: Extended drag and drop in action



You can drop items to this listbox:

Item no 107
Item no 69
Item no 70

Item no 89
Item no 90
Item no 91
Item no 92
Item no 108

You can drag lines from this listbox to the main listbox

Item no 63
Item no 64
Item no 65
Item no 66
Item no 67
Item no 68
Item no 69
Item no 70
Item no 71
Item no 72
Item no 73

Selected text can be dragged to the main listbox

Item no 86
Item no 87
Item no 88
Item no 89
Item no 90
Item no 91
Item no 92
Item no 93
Item no 94
Item no 95
Item no 96
Item no 97
Item no 98

Drag items from the listview to the main listbox

Item to select
Item no 107
Item no 108
Item no 109
Item no 110
Item no 111
Item no 112
Item no 113
Item no 114
Item no 115
Item no 116

```
  L:=LBMain.GetIndexAtY(Y);
  If L=-1 then
    begin
    L:=LBMain.Count-1;
    If L=-1 then
      L:=0;
    end;
  SDO:=Source as TStringsDragObject;
  For I:=SDO.Items.Count-1 downto 0 do
    LBMain.Items.Insert(L,SDO.Items[i]);
end;
```

The biggest part of this routine has in fact little to do with Drag & Drop: The position in the list of the drop is calculated (with some safety checks): that is where the items will be added. Note that for Delphi, the first line must be changed to

```
L:=LBMain.ItemAtPos(TPOint.Create(X,Y),true);
```

Once the position is known, all items from the `TStringDragObject` instance are copied over to the listbox. The final application (in Lazarus) can be seen in figure 3 on page 9.

6 Conclusion

This article has shown that Drag&Drop is really easy to implement be it from a file manager to a program or inside a program: It makes the operation of the program more intuitive for the user, so adding Drag&Drop should be considered by any Delphi or Lazarus programmer, when it is appropriate for the functioning of the program. One form of drag&drop has not yet been discussed, and that is dragging from one program to for example the file manager. Exploring this is left for a future contribution.