Advanced Drag & Drop

Michaël Van Canneyt

February 21, 2016

Abstract

In a previous contribution, we examined how simple drag and drop can be implemented for Delphi and Lazarus. In this article, we will look at a more complex example: drag and drop from an application to the windows explorer.

1 Introduction

In Delphi and Lazarus, it is fairly easy to implement Drag and Drop inside an application, and receiving files from the explorer is also easily implemented. It becomes more complicated if the task is to drag and drop files from an application into the windows explorer (or indeed any other application). Use cases can be very simple and varied:

- an FTP or browser application where the contents of a remote directory or website are shown, and the user should be able to drag selected files to a folder on the local machine.
- An email application where the mail and attachments are dragged to the explorer, where they should be saved as files.
- A list of pictures taken by a raspberry pi, still stored on the raspberry.

This kind of interaction with the explorer is somewhat more complicated to implement. Drag and drop to the explorer is implemented through the clipboard and makes use of several interfaces: These interfaces must be implemented by the application that wishes to implement drag and drop.

The concepts needed will be explained using a sample application which shows a simple listview, and the user can drag selected items from the listview to the explorer. The files in the listview do not actually exist, but are ‘virtual’ files.

2 Starting a Drag and Drop operation

To start a Drag & Drop operation to the explorer, the doDragDrop operation must be called:

Function DoDragDrop(dataObj: IDataObject;
dropSource: IDropSource;
dwOKEffects: Longint;
var dwEffect: Longint): HResult; stdcall;
This call involves 2 Interface parameters and 2 simple parameters. The dwOKEffects parameter gives an indication of what kind of operations are supported. It is an OR-ed combination of the following constants:

**DROPEFFECT_NONE** This value can only be returned in dwEffect: the drop target could not accept the data.

**DROPEFFECT_COPY** The data is copied to the target.

**DROPEFFECT_MOVE** The data is moved to the target (the source is responsible for deleting the data)

**DROPEFFECT_LINK** The data is linked in the target.

**DROPEFFECT_SCROLL** Scrolling is about to occur in the target. This value cannot be specified in DoDragDrop, but can be reported in IDropSource.

When the Drag & drop operation is finished, the DoDragDrop function will report the actual operation in dwEffect. The function will return S_OK if all went well.

The function needs also 2 interfaces: IDataObject and IDropSource. For the sample application, a single object will be made that implements these methods: TDnDObject. It will be used to implement the Drag And Drop in the sample application.

In practice, it might be a better choice to implement the interfaces in 2 separate objects. Since the object needs interfaces, we’ll descend it from TInterfacedObject, which will make sure it handles IUnknown and the associated reference counting mechanism.

### 3 Drag & Drop : Status reporting

The IDropSource interface is the simplest of the 2 interfaces involved in a drag & drop operation. It is used while the user is dragging to allow the source to change for example the appearance of the Drag&Drop cursor or to abort the operation. The methods of this interface are called continuously while the user is dragging.

It looks as follows:

```pascal
IDropSource = interface(IUnknown)
  function QueryContinueDrag(fEscapePressed: BOOL; grfKeyState: Longint): HResult;
  function GiveFeedback(dwEffect: Longint): HResult;
end;
```

The GiveFeedback allows the implementor to set the cursor or implement some other visual effects so the user is aware of what is happening. It can be implemented as follows:

```pascal
function TDnDObject.GiveFeedback(dwEffect: Integer): HResult;
begin
  // Maybe do something with dwEffect
  Result:=DRAGDROP_S_USEDEFAULTCURSORS;
end;
```

The return value of DRAGDROP_S_USEDEFAULTCURSORS tells Windows that it should display a default cursor for the current effect (one of copy, move, link etc.). If the application has set the cursor, then S_OK must be returned.
The `QueryContinueDrag` operation is called at regular intervals (when the mouse moves or relevant keyboard keys are pressed) to check if the operation must be aborted, or if a drop must be performed. It gets the current state of mouse and key buttons: `grfKeyState` contains a OR-ed combination of one of the following constants:

- **MK_CONTROL** The control key is pressed: this can be used to distinguish a copy or move operation.
- **MK_SHIFT** The shift key is pressed.
- **MK_ALT** The ALT key is pressed, this is normally used to indicate a shortcut is created.
- **MK_BUTTON** A mouse button is pressed.
- **MK_LBUTTON** The left mouse button is pressed.
- **MK_MBUTTON** The middle mouse button is pressed.
- **MK_RBUTTON** The right mouse button is pressed.

The `fEscapePressed` parameter speaks for itself; it will be true if the user has pressed escape, and wishes to abort the operation. (this can be used for example to cancel any downloads in progress).

The `QueryContinueDrag` function must return one of the following values:

- **S_OK** The drag and drop operation can continue normally.
- **DRAGDROP_S_DROP** The drag and drop operation must completed. This should be returned if the mouse button is released.
- **DRAGDROP_S_CANCEL** The drag and drop operation must be cancelled. This should be returned if the `fEscapePressed` parameter equals `True`. sed.

For our sample component, the code can be kept simple: as soon as the mouse button is released, the drag and drop should be completed, and if escape is pressed, it must be cancelled:

```pascal
function TDnDObject.QueryContinueDrag(fEscapePressed: BOOL; grfKeyState: Integer): HResult;
Var
  HaveMouseButton : Boolean;
begin
  HaveMouseButton := ((grfKeyState and MK_LBUTTON) <> 0) or ((grfKeyState and MK_RBUTTON) <> 0);
  if fEscapePressed and HaveMouseButton then Result := DRAGDROP_S_CANCEL
  else if not HaveMouseButton then Result := DRAGDROP_S_DROP
  else Result := S_OK;
end;
```
4 Drag & Drop : Data management

The first argument in the DoDragDrop is a IDataObject interface. It is used to exchange data between the source and target of the Drag and Drop operation. It is more complicated than the IDropSource interface, and is used both when implementing target and sources for a drag & drop operation.

Several kinds of data must be transferred between the source and target of the operation: in our case, the file names of files to be copied, and the actual file data.

The declaration of the IDataObject is as follows:

IDataObject = interface(IUnknown)
function GetData(const formatetcIn: TFormatEtc; out medium: TStgMedium): HResult;
function GetDataHere(const formatetc: TFormatEtc; var medium: TStgMedium): HResult;
function QueryGetData(const formatetc: TFormatEtc): HResult;
function GetCanonicalFormatEtc(const formatetc: TFormatEtc; out formatetcOut: TFormatEtc): HResult;
function SetData(const formatetc: TFormatEtc; var medium: TStgMedium; fRelease: BOOL): HResult;
function EnumFormatEtc(dwDirection: Longint; out enumFormatEtc: IEnumFormatEtc): HResult;
function DAdvise(const formatetc: TFormatEtc; advf: Longint; const advSink: IAdviseSink; out dwConnection: Longint): HResult;
function DUnadvise(dwConnection: Longint): HResult;
function EnumDAdvise(out enumAdvise: IEnumStatData): HResult;
end;

At least the following methods will need to be implemented:

EnumFormatEtc  this function is called to list the data made available by the drop source.
  This function returns again an interface which will be used to enumerate the available data.

QueryGetData  is an auxiliary function to determine whether GetData will return valid data for a certain format. The result of this function is advisory only.

GetData  this routine is called by the program accepting the drop to fetch the actual data.
  The actual data can come in many forms, but is one of the forms returned by EnumFormatEtc.

For the use case discussed here, the following methods are not needed:

GetCanonicalFormatEtc  Data can be transferred in various formats. This function can be called to see if one kind of format is equivalent to another for the drop source.

SetData  SetData needs to be implemented for Drop targets. Since we’re only looking at implementing a drop source, this interface does not need to be implemented.

DAdvise  is needed for advisory connections. This is an advanced feature which is not needed, and may be left unimplemented.

DUnadvise  is needed for advisory connections. This is an advanced feature which is not needed, and may be left unimplemented.
**EnumDAdvise** is needed for advisory connections. This is an advanced feature which is not needed, and may be left unimplemented.

These methods must of course be present in the TDnDObject since they are part of the IDataObject interface, but they can return a standard return value (E_NOTIMPL) for such cases:

```pascal
function TDnDObject.DAdvise(const formatetc: TFormatEtc; advf: Integer; const advSink: IAdviseSink; out dwConnection: Integer): HResult;
begin
  Result:=E_NOTIMPL;
end;

function TDnDObject.DUnadvise(dwConnection: Integer): HResult;
begin
  Result:=E_NOTIMPL;
end;

function TDnDObject.EnumDAdvise(out enumAdvise: IEnumStatData): HResult;
begin
  Result:=OLE_E_AdviseNotSupported;
end;

function TDnDObject.GetCanonicalFormatEtc(const formatetc: TFormatEtc; out formatetcOut: TFormatEtc): HResult;
begin
  FillChar(formatetcOut,SizeOf(TFormatEtc),0);
  Result:=E_NOTIMPL;
end;

function TDnDObject.GetDataHere(const formatetc: TFormatEtc; out medium: TStgMedium): HResult;
begin
  FillChar(medium,SizeOf(TStgMedium),0);
  Result:=E_NOTIMPL;
end;

function TDnDObject.SetData(const formatetc: TFormatEtc; var medium: TStgMedium; fRelease: BOOL): HResult;
begin
  FillChar(medium,SizeOf(TStgMedium),0);
  Result:=E_NOTIMPL;
end;
```

5 Drag & Drop : Data formats

Data in a drag and drop operation is transferred in one of the formats that the shell (explorer) clipboard understands. This data then must be transferred in one of the ways that the shell understands.
The shell clipboard supports many data formats, they are listed at (the following should be one line):


The ones interesting for the use case discussed are:

**CF_HDROP** This data format signifies existing files in the system.

**CFSTR_FILECONTENTS** This data format must be used to transfer actual file contents.

**CFSTR_FILEDESCRIPTOR** This data format must be used to transfer a series of file descriptors.

**CFSTR_FILENAME** This data format can be used to transfer a single filename.

**CFSTR_FILENAMEMAP** This data format can be used to transfer a series of file names.

**CFSTR_MOUNTEDVOLUME** This data format is used to indicate a mounted volume (a disk).

**CFSTR_SHELLIDLIST** This data format can be used to transfer a series of files, but the files are identified by a file system identifier.

**CFSTR_SHELLIDLISTOFFSET** This data format is used to transfer the location (on screen) of the transferred objects.

If a virtual file needs to be transferred, 2 data formats must be used together:

**CFSTR_FILEDESCRIPTOR** This format must be used to transfer a series of file descriptors: this is a list of records that describe the files involved in the drag and drop operation. The files must not actually exist, but they can be described by the records in this format.

**CFSTR_FILECONTENTS** This format must be used to transfer the actual file contents.

The various CFSTR_* formats must be registered with the OLE mechanism. For each registered format, the OLE mechanism will return an numerical identifier (CF_HDROP is such a fixed system identifier). Since our application will need the above 2 formats, they are registered in the initialization section of the unit, and the numerical values will be saved for later use in the code.

```
begin
  CF_Names:=RegisterClipBoardFormat(CFSTR_FILEDESCRIPTOR);
  CF_Contents:=RegisterClipBoardFormat(CFSTR_FILECONTENTS);
end.
```

The CF_Names and CF_Contents variables will be used further in the code.

The above describes what is being transferred. It does not describe how the data will be transferred. The transfer mechanism is described using several constants:

**TYMED_HGLOBAL** Data is transferred in a globally allocated memory block.

**TYMED_FILE** Data is transferred using a filename of an existing file on disk.

**TYMED_ISTREAM** Data is transferred using the IStream streaming mechanism.
TYMED_IStorage  Data is transferred using the IStorage storage component mechanism.

TYMED_GDI  Data is transferred using a GDI object: a bitmap.

TYMED_MFPICT  Data is transferred using a GDI metafile.

TYMED_ENHMIF  Data is transferred using a GDI enhanced metafile.

Obviously, not all types of transfer can be used for all kinds of data format. When a drag&drop source enumerates the data it can deliver, it must for each kind of available data indicate which transfer mechanism is available. The above constants can be OR-ed together to indicate the available mechanisms. When data is actually transferred, the used mechanism must be indicated using one of the above constants.

To know what data is available from a drag source, or what data a drop target accepts, the drag&drop loop calls EnumFormatEtc. This function needs to return an interface which will be used to enumerate the available or acceptable data. The dwDirection parameter determines whether the data is fetched or pushed. For the sample application, this means we only need to implement the read direction (DATADIR_GET):

```pascal
function TDnDObject.EnumFormatEtc(dwDirection: Integer; out enumFormatEtc: IEnumFormatEtc): HResult;
begin
  if (dwDirection=DATADIR_GET) then
  begin
    EnumFormatEtc:=TEnumFormatEtc.Create;
    Result:=S_OK;
  end
  else
  begin
    EnumFormatEtc:=nil;
    Result:=E_NOTIMPL;
  end;
end;
```

The TEnumFormatEtc class implements the IEnumFormatEtc interface:

```pascal
TEnumFormatEtc = class(TInterfacedObject, IEnumFormatEtc)
  FIndex : Integer;
public
  function Next(celt: Longint; out elt; pceltFetched: PLongint): HResult;
  function Skip(celt: Longint): HResult;
  function Reset: HResult;
  function Clone(out Enum: IEnumFormatEtc): HResult;
end;
```

The methods of this interface resemble the standard methods found in an enumerator class as found in Delphi.

The most important method is the Next method; It receives a maximum number (celts) of TFormatEtc elements to retrieve, and must fill the elts memory block with as much elements as available. If pceltFetched is not nil, the number of actually fetched elements must be written there. This function can be called multiple times, till all available formats have been fetched.
The TFormatEtc record describes the available data and transfer mechanisms, and is declared as follows:

```pascal
TFORMATETC = record
  cfFormat: TClipFormat;
  ptd: PDVTargetDevice;
  dwAspect: Longint;
  lindex: Longint;
  tymed: Longint;
end;
```

The various fields have the following meaning:

- **cfFormat** The available clipboard format. This is one of the standard clipboard formats, or a registered clipboard format.
- **ptd** A target device descriptor. This is not needed for our implementation.
- **dwAspect** A rendering aspect (how much detail is allowed in the rendering) For the purpose of file copying, this is always DVASPECT_CONTENT
- **lindex** Usually -1, to indicate all of the data must be rendered. If not -1, it is a zero-based index of the part to be rendered. In the case of copying multiple files, this will indicate which file in the list must be copied.
- **tymed** The types of storage medium used to transfer data.

For our implementation, 2 kinds of data must be reported: the filenames (clipboard format CF_NAMES) and the file contents (CF_Contents). The Findex field is kept to know at what position the enumerator currently is.

```pascal
function TEnumFormatEtc.Next(celt: Integer; 
  out elt; 
  pceltFetched: PLongint): HResult;
```

Var

```
CC,C : Integer;
F : TFormatEtc;
P : ^TFormatEtc;
```

begin

Result:=S_False;
P:=Elt;
CC:=0; // Copied count.
// This data is the same for all records.
F.ptd:=nil;
F.dwAspect:=DVASPECT_CONTENT;
F.lindex:=-1;
// Valid indexes are 0 and 1
While (FIndex<2) and (CElt>0) do
begin
  Inc(CC);
  Case FIndex of
  0 :
    begin // File names are available.
```
The loop transfers the data to the elts memory block, taking care only to report as much data as still is available. After the loop has finished, the result is determined and the number of elements is returned to the caller.

The clone method needs to return a copy of the enumerator, which preserves the state:

```pascal
function TEnumFormatEtc.Clone(out Enum: IEnumFormatEtc): HResult;
begin
  Result:=S_OK;
  try
    Enum:=TEnumFormatEtc.Create;
    TEnumFormatEtc(Enum).FIndex := FIndex;
    except
      Result:=E_Fail;
    end;
  end;
end;
```

The method reset simply resets the enumerator:

```pascal
function TEnumFormatEtc.Reset: HResult;
begin
  FIndex:=0;
  Result:=S_OK;
end;
```

And the Skip method just increments the position of the enumerator:

```pascal
function TEnumFormatEtc.Skip(ceI: Integer): HResult;
begin
  if FIndex+Celt<=2 then
    begin
      inc(FIndex,Celt);
      Result:=S_OK;
    end
  else
```
6 Drag & Drop: transferring data

Now that we've reported what data is available, the last part of our object must be implemented: actually transferring data. This must done by implementing the GetData method. The caller passes a description of the needed data in the formatetcIn, and the routine must respond by returning the requested data and passing it back through medium.

For the file drag and drop mechanism, the method needs to return 2 kinds of data: filenames of files to copy (using descriptors) and for each of the files, the file contents. In the case of file contents, the lIndex field of the formatetc record will contain the index of the file that must be returned. This means that the implementation can be done as follows:

```pascal
function TDnDObject.GetData(const formatetcIn: TFormatEtc; out medium: TStgMedium): HResult;
begin
    Result:=E_FAIL;
    if (FormatEtcIn.cfFormat=cf_names) then
        Result:=GetFileNames(medium)
    else if (FormatEtcIn.cfFormat=cf_contents) then
        Result:=GetFileContents(formatetcin.lindex,medium);
end;
```

Since we wish to be able to drag and drop multiple files from our application, the list of (virtual, they need not actually exist) filenames is passed to the constructor of our TDnDObject, and the list is freed when the object is freed:

```pascal
constructor TDnDObject.Create(AFiles : TStrings);
begin
    FFiles:=AFiles;
end;
```

```pascal
der destructor TDnDObject.Destroy;
begin
    FreeAndNil(FFiles);
    inherited;
end;
```

Using this list, the GetFileNames routine can be implemented as follows. It starts out by reserving a block of memory for a TFileGroupDescriptorW structure, which is in essence an array of TFileGroupDescriptorW records, preceded by a count.

```pascal
function TDnDObject.GetFileNames(out medium: TStgMedium): HResult;
Var
    G : TFileGroupDescriptorW;
    H : HGlobal;
    Psrc : PFileGroupDescriptorW;
    PF : PFileDescriptorW;
    FN : String;
begin
    // Reserve block of memory
    H := AllocMem(HGlobal, sizeof(TFileGroupDescriptorW) + sizeof(TFileDescriptorW) * FFiles.Count);
    // Initialize TFileGroupDescriptorW
    G := (PFileGroupDescriptorW(H))^;
    G.Size := sizeof(TFileGroupDescriptorW) + sizeof(TFileDescriptorW) * FFiles.Count;
    G.cFiles := FFiles.Count;
    G.pFirst := (PFileDescriptorW(H))^;
    for FN := FFiles[0] to FFiles[FFiles.Count-1] do
        G.pFirst := GD;
        GD := GD.Next;
        GD.Name := FN;
    // Return block of memory
    medium := TStgMedium(H);
end;
```
P : PByte;
I, Count, Len : Integer;

begin
  Count := FFiles.Count;
  // length of the block.
  Len := SizeOf(TFileGroupDescriptorW) +
        SizeOf(TFileDescriptorW) * (Count - 1);
  GetMem(PSrc, Len);
  FillChar(PSrc^, Len, 0);
  PSrc^.cItems := Count;

  The implementation then proceeds to copy all filenames to the array of file descriptor records:

  PF := @PSrc^.fgd[0];
  {$R-} // Eliminate a range error.
  for I := 0 to Count - 1 do
    begin
      FN := FFiles[i];
      Move(FN[1], PF[i].cFileName, Length(FN) * SizeOf(Char));
    end;

  Note that the TFileDescriptorW record contains many other fields besides the cFileName field, such as size, date of access etc. These fields are not needed for our implementation, so they are ignored.

  The implementation then copies the structure to a global memory block, which is then returned in the medium record using the TYMED_HGLOBAL type:

  H := GlobalAlloc(GHND or GMEM_SHARE, Len);
  P := GlobalLock(H);
  try
    Move(PSrc^, P^, Len);
  finally
    FreeMem(PSrc);
    GlobalUnlock(H);
  end;
  with Medium do
    begin
      tymed := TYMED_HGLOBAL;
      hGlobal := H;
      unkForRelease := nil;
    end;
  Result := S_OK;
end;

The last thing that remains to be done is the GetFileContents routine, which must transfer the actual file data. This is done using an event handler of the TDNDObject class:

Type
  TGetStreamEvent = Procedure (Sender : TObject;
                               Const AFileName : String;
                               out S : TStream) of object;
TDnDObject = class(TInterfacedObject, IDataObject, IDropSource)
public
  constructor Create(AFiles : TStrings);
  destructor Destroy; override;
  // Property OnGetStream
  Property OnGetStream : TGetStreamEvent;
end;

The event handler will be called for each file that needs to be copied.

Using this event handler, the GetFileContents method is fairly easy. It gets the stream through the event handler, wraps it in a TStreamAdaptor class and returns the resulting IStream interface through the medium record:

function TDnDObject.GetFileContents(const Index: Integer; out medium: TStgMedium): HRESULT;
Var
  S : TStream;
begin
  with Medium do
  begin
    Tymed:=TYMED_ISTREAM;
    S:=Nil;
    if Assigned(FOnGetStream) then
      FOnGetStream(Self,FFiles[Index],S);
    S.Position:=0;
    IStream(stm):=TStreamAdapter.Create(S,soOwned) as IStream;
    UnkForRelease:=nil;
    Result:=S_OK;
  end;
end;

The Drag and Drop mechanism is responsible for the release of the IStream interface. The StreamAdapter will then free the stream S, because of the soOwned parameter passed in the constructor.

With this last method implemented, the TDnDObject class is ready to be used.

7 Putting it all together

To use the TDnDObject class, we create a form with a listview (LVfiles) in it, and add some virtual items to the listview. The drag and drop operation is started manually, as the automatic drag and drop will not work for drops outside the application.

The drag and drop detection is started with a OnMouseDown event handler in the listview:

procedure TForm1.LVFilesMouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button=mbLeft then
begin
FStartX:=X;
FStartY:=Y;
end
else
begin
FStartX:=-1;
FStartY:=-1;
end;
end;

A drag and drop is not started, unless the user has dragged the mouse at least 10 pixels. In that case, the selected (checked) items are collected in a stringlist, and this is passed on to a newly created TDNDObject instance. This instance is then passed on to the DoDragDrop call, which sets the ball rolling for the Drag and Drop operation.

procedure TForm1.LVFilesMouseMove(Sender: TObject;
   Shift: TShiftState;
   X,Y: Integer);
Var
   DND : TDNDObject;
   I,Eff : Integer;
   AFiles : TStrings;
begin
if (ssLeft in Shift) and (FStartX>0)
   and ((Abs(X-FStartX)>10) or (Abs(Y-FStartY)>10)) then
begin
   // Collect checked files
   AFiles:=TStringList.Create;
   for I:=0 to LVFiles.Items.Count-1 do
      If LVFiles.Items[I].Checked then
      AFiles.Add(LVFiles.Items[I].Caption);
   DND:=TDNDObject.Create(AFiles);
   DND.OnGetStream:=DoGetStream;
   DoDragDrop(DND as IDataObject,DND as IDropSource,DropEffect_Copy,eff);
end;
end;

As can be seen, the only allowed effect is copy DROPEFFECT_COPY. The OnGetStream event handler, which will create the actual file contents, is set to a DoGetStream method. This method is very simple:

procedure TForm1.DoGetStream(Sender : TObject;
   Const AFileName : String;
   Out S : TStream);
begin
   S:=TStringStream.Create(’This is some faked data for file “’+AFileName+’”
end;

In a real-world application, this method would download a file or otherwise create file contents, and would then pass the contents in a stream. Here some dummy data is created.
Figure 1: Drag and Drop in action

Note that the passed filename is a virtual file, which doesn’t actually exist on the system. It should be sufficient to determine which data is to be generated.

The result of all this is displayed in figure 1 on page 14.

8 Conclusion

Implementing drag and drop to the explorer manually is not for the faint of heart, but can be done, as demonstrated in this article.

The sample code presented here can be modified to suit various use cases where an application acts as a drag and drop source. A more complete suite of components which makes life easier, can be found on GitHub:


Exploring the workings of this component suite is out of scope for this article but will be left for a future contribution.