Drag and drop Part 3

Michaël Van Canneyt

December 27, 2021

Abstract

In two previous articles the principles behind drag and drop in windows were explored, especially drag and drop to the explorer. In this article drag and drop is explored even further: Support for dragging filenames to other applications is added, a set of components to handle drag and drop for a control is introduced, and finally drops in a Delphi application are explored.

1 Introduction

In previous articles simple drag and drop inside a Delphi application was explored. This was expanded to dragging (virtual) files to the explorer. The code presented in that article had a drawback: it worked only for the explorer or applications that can handle advanced drag and drop mechanisms:

Dragging files to editors such as Notepad++ would not work with the mechanism explained in the second article of this series. Dragging files to another Delphi application - which is using the mechanism explained in the first article - also will not work. The code presented in this article will remedy that.

To implement dragging or dropping, custom code was needed: code to detect the start of a drag operation. This code can be collected in a `TComponent` descendant which can be dropped on a form, coupled to a visual control: this exercise was done by Peter van der Sman, and his work will be expanded upon here.

Lastly, we'll implement the drop side of inter-application drag and drop: this will allow applications to receive file contents (indeed, any contents) from another application, provided it understands the format used by the drag source.

2 Implementing CF_HDROP to report filenames

As explained in the previous article, there are several clipboard formats that can be used to transfer filenames. In the previous article, 2 clipboard formats were used to transfer filenames and file contents, these formats were registered in the initialization section of the dnd unit:

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

The article also explained that the `CF_HDROP` clipboard format could be used to report (existing) filenames. This is the clipboard format that is transformed by Windows to a

WM_DROPFILES message if the target application does not handle Drag & Drop explicitly: the first article explained how to deal with this message. Capturing this message is how most applications handle dropping of files.

So, how to expand the code so the WM_DROPFILES message can be generated ? This is quite easy, and consists of 2 parts: The first is to announce the availability of the CF_HDROP clipboard format in the TEnumFormatEtc class. For this, we extend the constructor with a parameter:

```
constructor TEnumFormatEtc.Create(AReportCFHDrop: Boolean);
beginM
  FReportCFHDrop:=AReportCFHDrop;
end;
```

The clone method of the class copies this field. The Next method in the enumerator is adapted so it reports the format. The CF_HDROP format must be made available with a media type of TYMED_HGLOBAL:

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

Var
  Max,CC,C : Integer;
  F : TFormatEtc;
  P : ^TFormatEtc;

begin
  P:=@Elt;
  CC:=0;
  // Valid indexes are 0 and 1 or 2 if FReportCFHDrop is true
  Max:=2+Ord(FReportCFHDrop);
  While (FIndex<Max) and (CElt>0) do
    begin
    Inc(CC);
    Case FIndex of
    0 :
       begin
       F.cfFormat:=cf_names;
       F.ptd:=nil;
       F.dwAspect:=DVASPECT_CONTENT;
       F.lindex:=-1;
       F.tymed:=TYMED_HGLOBAL;
      end;
    1 :
      begin
      F.cfFormat:=cf_contents;
      F.ptd:=nil;
      F.dwAspect:=DVASPECT_CONTENT;
      F.lindex:=-1;
      F.tymed:=TYMED_ISTREAM;
      end;
    2 :  // This code is new
       begin
       F.cfFormat:=cf_Hdrop;
```

```
        F.ptd:=nil;
        F.dwAspect:=DVASPECT_CONTENT;
        F.lindex:=-1;
        F.tymed:=TYMED_HGLOBAL;
       end;
    end;
    Move(F,P^,SizeOf(TFormatEtc));
    Dec(Celt);
    Inc(FIndex);
    end;
  Result:=S_False;
  if pceltFetched<>nil then
    pceltFetched^:=CC;
  if (CC>0) then
    Result:=S_OK;
end;
```

And that is all that there is to it. With this code, files can be dragged to NotePad++ or any application that responds to the `WM_DROPFILES` message.

3 A design-time component to handle dragging operations

Initiating a drag&drop to another application is manual work. This can be reduced by creating a simple component which can be dropped on a form:

```
TDragFilesHandler=class(TCustomDragFilesHandler)
published
  Property WinControl;
  Property Active;
  Property FileNames;
  property DragTreshold ;
  property OnGetFilesToDrag;
  property OnGetStream ;
end;
```

These properties are quite self-explanatory:

WinControl is the control which should be drag-enabled.

Active Controls whether a dragging operation is allowed or not.

FileNames is the list of filenames to report.

DragTreshold is the number of pixels the mouse must move before a drag operation is started.

OnGetFilesToDrag is an event that is called when a drag is started, it can be used to fill the filenames property with the names of files to drag.

OnGetStream Is called when a drop occurred and the contents of a file must be fetched.

This component is quite simple, and most of the code in it is needed to correctly handle designing in the IDE. The central pieces are the following. When dragging is enabled, the following code is executed:

```
procedure TCustomDragFilesHandler.StartDragging;
begin
  StopDragging;
  FSavedMouseDown:=TMyControl(FControl).OnMouseDown;
  FSavedMouseMove:=TMyControl(FControl).OnMouseMove;
  TMyControl(FControl).OnMouseDown:=ControlMouseDown;
  TMyControl(FControl).OnMouseMove:=ControlMouseMove;
end;
```

Any existing `OnMouseDown` and `OnMouseMove` handlers are saved and replaced by custom versions, where the `ControlMouseDown` method is used to detect the start of the drag operation:

```
procedure TCustomDragFilesHandler.ControlMouseDown(
    Sender:TObject;
    aButton: TMouseButton;
    aShift: TShiftState;
    aX,aY: Integer);

begin
  if Factive then
    begin
    FDragStarted:=(aButton=mbLeft);
    if FDragStarted then
      begin
      FDragStart.X:=aX;
      FDragStart.Y:=aY;
      end;
    end;
  if assigned(FSavedMouseDown) then
    FSavedMouseDown(Sender,aButton,aShift,aX,aY);
end;
```

Note that the original OnMouseDown handler is called at the end of the routine. Similarly, the OnMouseMove handler detects the start of a drag operation, and creates an instance of the `TDNDObject` to handle the Drag&Drop operation. The working of that class was demonstrated in the previous article.

To detect the start of a drag, we check if the mouse has moved a certain distance with the left mouse button pressed:

```
procedure TCustomDragFilesHandler.ControlMouseMove(
    Sender:TObject;
    aShift: TShiftState;
    aX,aY: Integer);

Var
  lDND     : TDNDObject;
  lEffect  : Integer;
  L : TStringList;

begin
  if FDragStarted and (ssLeft in aShift) then
    if (Abs(aX-FDragStart.X)>FDragTreshold)
        or (Abs(aY-FDragStart.Y)>FDragTreshold) then
```

```
    begin
    FDragStarted:=False;
    FOnGetFilesToDrag(self);
    if FFileNames.Count>0 then
      begin
      L:=TStringList.Create;
      lDND:=TDNDObject.Create(L);
      L.Assign(FFileNames);
      lDND.OnGetStream:=FOnGetDragStream;
      ActiveX.DoDragDrop(lDND as IDataObject,
                         lDND as IDropSource,
                         DropEffect_Copy,lEffect);
      end;
    end;
  if assigned(FSavedMouseMove)then
    FSavedMouseMove(Sender,aShift,aX,aY);
end;
```

That's all there is to it. The `TDragFilesHandler` component can be registered in the IDE and dropped on a form to – with the help of some mouse clicks – enable dragging from a wincontrol on the form .

4 Handling drops

Now that implementing dragging holds no more secrets, it is time to go to the other end of the communication line, and handle drops. There are 2 mechanisms. The first one was explored in the first article in this series: handling the `WM_DROPFILES` message. This works for existing filenames.

Handling all other kind of dropped data is also quite simple. To handle drops for a window, it is necessary to register a `IDropTarget` interface for the wincontrol instance that must handle the drop. The Registering and unregistering the `IDropTarget` interface for a window happens with the following two functions:

```
Function RegisterDragDrop(wnd: HWnd;
                          dropTarget: IDropTarget): HResult;
Function RevokeDragDrop(wnd: HWnd): HResult;
```

These functions are available through the Winapi.ActiveX unit.

The `IDropTarget` interface looks as follows:

```
IDropTarget = interface(IUnknown)
function DragEnter(const dataObj: IDataObject;
                   grfKeyState: Longint; pt: TPoint;
                   var dwEffect: Longint): HResult;
function DragOver(grfKeyState: Longint; pt: TPoint;
                  var dwEffect: Longint): HResult;
function DragLeave: HResult;
function Drop(const dataObj: IDataObject;
              grfKeyState: Longint; pt: TPoint;
              var dwEffect: Longint): HResult;
end;
```

The methods each have their purpose:

DragEnter is called when a drag operation enters the control. The `IDataObject` interface that was created by the drag source is passed to the method, together with some other data. The method must return the drag&drop method that will be executed: none, copy, move or link. The methods of `IDataObject` can be used to get information on what is being dragged.

DragLeave is called when a drag operation leaves the control. Any data allocated during the DragEnter can be released here.

DragOver is called multiple times, as long as the dragging operation is moving over the control.

Drop is called when a drop is performed on the control. Again the `IDataObject` interface that was created by the drag source is passed to the method, its methods can be used to get the data from the drag source.

The class that we will create to implement the `IDropTarget` interface is called `TDropObject`. It handles the drag and drop exclusively using events as follows:

- When a drag operation enters the control, the methods of `IDataObject` and the format enumerator are used to retrieve a list of formats offered by the drag source. An event handler is then called to see if the application can handle one of these formats. The application must respond with one of the drop effects: none, copy move or link.
- The `DragOver` operation is simply passed on to an event handler: the application can set this event handler and can react as is appropriate for its functioning.
- Similarly, the `DropLeave` operation is simply passed on to an event handler.
- The `Drop` method again starts by requesting the available formats from the drag source. It then calls an event handler (OnGetDropFormat) in a loop (iterations) : the event handler must indicate which of the available formats it is prepared to handle, and how often this format must be requested.
  For each of these formats, the method will then start a second loop (items) in which it fetches the data from the drag source, and presents it to the application with a second event handler (`OnDropData`).

This double loop mechanism is necessary: to fetch file data from a drag source, 2 steps are needed:

1. The first iteration is to get the list of filenames that are being dropped.
2. The second iteration fetches the data for each of the filenames received in step 1.

The declaration of the object that will do all this is as follows:

```
TDropObject = class(TInterfacedObject, IDropTarget)
  // Methods, Properties
  Procedure SetHandle(AHandle : THandle);
  Procedure ClearHandle;
  procedure DropFileNamesToStringList(S: TStream; AFiles: TStrings);
  procedure FileDescriptorToStringList(S: TStream; AFiles: TStrings);
  Property FormatCount : Integer;
  Property Formats[AIndex : Integer] : TFormatDescription;
  // Events
  Property OnLeave : TNotifyEvent;
```

```
  Property OnDrop : TDataDropEvent;
  Property OnEnter : TFileDragEnterEvent;
  Property OnOver : TFileDragOverEvent;
  Property OnGetDropFormat : TDropFormatEvent;
  Property OnDropData : TDropDataEvent;
end;
```

The `SetHandle` and `ClearHandle` set and clear the windows handle for which drops must be accepted. These methods call `RegisterDragDrop` and `RevokeDragDrop` for the handle: these methods signal to Windows that the control accepts drops (or no longer accepts them). They also increase and decrese the reference count of the interface, so that the object is not accidentally freed by Windows.

```
Procedure TDropObject.SetHandle(AHandle: THandle);
begin
  FHandle:=AHandle;
  RegisterDragDrop(FHandle,Self As IDropTarget);
  _AddRef;
end;

procedure TDropObject.ClearHandle;
begin
  RevokeDragDrop(FHandle);
  _Release;
end;
```

The start of everything is the `DragEnter` method, which is quite simple

```
function TDropObject.DragEnter(const dataObj: IDataObject;
    grfKeyState: Integer;  pt: TPoint;
    var dwEffect: Integer): HResult;

Var
  Count : Integer;

begin
  // Little point in continuing if no handlers are set.
  if Not Assigned(Self.OnEnter) and Not Assigned(Self.OnDrop) then
    begin
    dwEffect:=DROPEFFECT_NONE;
    Result:=S_OK;
    Exit;
    end;
  Count:=GetFormats(DataObj);
  // if we don't understand the media, little point in continuing
  if Count=0 then
    begin
    dwEffect:=DROPEFFECT_NONE;
    Result:=S_OK;
    Exit;
    end;
  dwEffect:=DROPEFFECT_COPY;
  If Assigned(FOnEnter) then
    FOnEnter(Self,FAvailableFormats,pt.X,pt.Y,dwEffect);
```

```
  Result:=S_OK;
end;
```

The bulk of the work is done by the `GetFormats` method. This method will query the `IDataObject` interface to get a list of clipboard formats offered. Only formats that work with an `IStream` interface or `HGLobal` global memory handle will be retrieved, because these are the only ways of retrieving data that the class has implemented.

The `GetFormats` method returns the number of retrieved clipboard formats. if no supported formats are offered, the method exits. The available formats are kept in a dynamic array:

```
TFormatDescription = Record
  MediaType : Integer;
  ClipBoardFormat : Integer;
  FormatName : String;
end;
TFormatDescriptionArray = Array of TFormatDescription;
```

This array is passed on to the `OnEnter` method: the method should return the drop operation that will be performed: `DROPEFFECT_COPY` is set as the default. If the application does not understand the format, `DROPEFFECT_NONE` can be used to indicate that dropping is not allowed.

The `GetFormats` method uses the enumerator interface (`IEnumFORMATETC`) which it gets from the `IDataObject` interface to retrieve the available formats. It does this in 2 steps. The first is to collect the formats which use `TYMED_HGLOBAL` or `TYMED_ISTREAM` to transfer data:

```
function TDropObject.GetFormats(const dataObj: IDataObject): Integer;

Var
  N : Array[0..MaxBufSize] of Char;
  F,M : Array[0..MaxFormatCount] of Integer;
  MediaOK : Boolean;
  Enum : IEnumFORMATETC;
  Elt : TFormatEtc;
  I,E : longint;
  S : String;

begin
  Result:=0;
  SetLength(FAvailableFormats,0);
  if DataObj.EnumFormatEtc(DATADIR_GET,Enum)<>S_OK then
    exit;
  MediaOK:=False;
  Repeat
    I:=0;
    if Enum.Next(1,elt,@I)=S_OK then
      begin
      if (Elt.tymed=TYMED_HGLOBAL)
         or (elt.tymed=TYMED_ISTREAM) then
        begin
        F[Result]:=elt.cfFormat;
        M[Result]:=elt.tymed;
```

```
        Inc(Result);
        MediaOK:=True;
        end;
      end;
  Until I<>1;
  Enum:=Nil; // Release instance
```

Note that at the end of this part, `Result` contains the number of available formats. In the second part the numerical formats are transformed to human-readable names using the `GetClipboardFormatName` windows API call:

```
 if Result>0 then
    begin
    SetLength(FAvailableFormats,Result);
    For I:=0 to Result-1 do
      begin
      if F[I]<=CF_MAX then
        S:=Predefined[F[I]]
      else
        begin
        FillChar(N[0],SizeOf(N),#0);
        If 0=GetClipboardFormatName(F[I],@N,MaxBufSize) then
          S:='Predefined format '+IntToStr(F[i])
        else
          S:=N;
        end;
      FAvailableFormats[I].ClipBoardFormat:=F[i];
      FAvailableFormats[I].MediaType:=M[i];
      FAvailableFormats[I].FormatName:=S;
      end;
    end;
end;
```

Some clipboard formats (with numerical IDs smaller than `CF_MAX`) are pre-defined, these names are fetched from an array of constants, as windows will not report their names.

The `DragOver` and `DragLeave` methods are so simple, they do not need explaining:

```
function TDropObject.DragLeave: HResult;
begin
  if Assigned(FOnLeave) then
    FOnLeave(Self);
  Result:=S_OK;
end;

function TDropObject.DragOver(grfKeyState: Integer;
  pt: TPoint;
  var dwEffect: Integer): HResult;

begin
  if Assigned(FOnOver) then
    FOnOver(Self,grfKeyState,pt.X,pt.Y,dwEffect)
  else
    dwEffect:=DROPEFFECT_COPY;
  Result:=S_OK;
```

```
end;
```

The `Drop` method will implement the double loop that was discussed above. It starts again by getting the list of formats:

```
function TDropObject.Drop(const dataObj: IDataObject;
                          grfKeyState: Integer;
                          pt: TPoint;
                          var dwEffect: Integer): HResult;

Var
  UseFormat,ACount,AListCount : Integer;

begin
  ACount:=GetFormats(dataObj);
  if (ACount=0) or ((ACount>1) and Not Assigned(FonDropFormat)) then
    begin
    dwEffect:=DROPEFFECT_NONE;
    Result:=s_OK;
    Exit;
    end;
  ACount:=0;
```

And now the double loop starts. It starts by calling the `FOnDropFormat` handler to see what format must be retrieved. If there is no handler, and there is only 1 available format, it is selected automatically:

```
  Repeat
    UseFormat:=-1;
    AListCount:=0;
    // Ask what format we need. Pass on the iteration (ACount).
    If Assigned(FOnDropFormat) then
      FOnDropFormat(Self,FAvailableFormats,ACount,UseFormat,AListCount);
    else if Length(FAvailableFormats)=1 then
      UseFormat:=0; // No need to ask.
    // If a format was chosen, fetch it:
    if (UseFormat>=0) then
      if (Not FetchClipBoardData(DataObj,ACount,UseFormat,AListCount))
        or (Length(FAvailableFormats)=1) then
        UseFormat:=-1
      else
        Inc(ACount);
  Until (UseFormat<0) or (UseFormat>=Length(FAvailableFormats));
  if ACount=0 then
    dwEffect:=DROPEFFECT_NONE
  else
    dwEffect:=DROPEFFECT_COPY;
  Result:=s_OK;
end;
```

If a format was selected, the data is fetched and passed on to the application. This happens in `FetchClipBoardData`. The loop ends if no clipboard format was selected, or if there was only a single format available. Note that `AListCount` is the number of iterations that must be done when fetching the data: it is set to zero (it is a zero-based index). It can be set by the application.

The FetchClipBoardData method is responsible for fetching data from the clipboard and presenting it to the application in the form of a stream. It calls IDataObject.GetData with the appropriate parameters: a TagFormatEtc record that describes what data is wanted, and in return gets a TagSTGMedium record which contains the necessary fields to retrieve the data.

```
function TDropObject.FetchClipBoardData(
    const dataObj: IDataObject;
    Iteration, AFormat, AListCount : Integer) : Boolean;

Var
  Fmt : TagFormatEtc;
  I : INteger;
  medium : TagSTGMedium;
  S : TStream;

begin
  Result:=True;
  For I:=0 to AListCount do
    begin
    // Construct TagFormatEtc for GetData call:
    fmt.cfFormat:=FAvailableFormats[AFormat].ClipBoardFormat;
    fmt.tymed:=FAvailableFormats[AFormat].MediaType;
    fmt.dwAspect:=DVASPECT_CONTENT;
    fmt.lindex:=i; // Loop counter
    fmt.ptd:=Nil;
    // Now callGetData.
    if DataObj.GetData(fmt,medium)=S_OK then
      begin
      S:=TMemoryStream.Create;
      // Transform to stream
      Case medium.tymed of
        TYMED_HGLOBAL : GetMemStream(medium.hGlobal,S);
        TYMED_ISTREAM : GetStreamStream(IStream(medium.stm),S);
      end;
      S.Position:=0;
      // Pass on to application.
      if Assigned(FOnDropData) then
        FOnDropData(Self,Iteration,AFormat,I,S);
      // If the application didn't free the stream, we do:
      if Assigned(S) then
        FreeAndNil(S);
      end;
    end;
end;
```

The GetMemStream and GetStreamStream methods fetch the data through an appropriate mechanism. For a TYMED_HGLOBAL, this is done using a simple read from a memory location:

```
Procedure TDropObject.GetMemStream(H : THandle; S : TStream);

Var
  P : PByte;
```

```
begin
  // Get a pointer to the memory block.
  P:=GlobalLock(H);
  try
    // Read block data
    S.WriteBuffer(P^,GlobalSize(H));
  finally
    // Free & release memory block.
    GlobalUnlock(H);
    GlobalFree(H);
  end;
end;
```

The `GetStreamStream` method uses a `TStreamAdapter` class and uses the `IStream.CopyTo` method to fetch the data from the drag source:

```
Procedure TDropObject.GetStreamStream(Instream: IStream; S : TStream);

Var
  A : TStreamAdapter;
  stg : tagSTATSTG;
  R,W : UINT64;

begin
  // Create adapter
  A:=TStreamAdapter.Create(S);
  // Get size, and copy size bytes.
  if InStream.Stat(stg,STATFLAG_NONAME)=S_OK then
    Instream.CopyTo(A as IStream ,stg.cbSize,R,W);
end;
```

The adapter instance is automatically freed after the call to `CopyTo`, since it is a reference counted interface.

5 A Component wrapper around the TDropObject

The `TDropObject` class presented above cannot be dropped on a form. It is quite low-level: it accepts a window handle instead of a `TWinControl` instance. As in the case of the `TDndObject` class, a component can be made that takes care of the low-level code. We'll name this component `TDropFilesHandler`:

```
TDropFilesHandler=class(TCustomDropFilesHandler)
published
  Property WinControl;
  Property Active;
  Property FileNamesOnly;
  property OnDropFile;
  Property OnLeave;
  Property OnEnter;
  Property OnOver;
  Property OnGetDropFormat;
  Property OnDropData;
```

```
end;
```

Most of the event properties speak for themselves, they are the same as for the `TDropObject` class. The remaining properties are also fairly obvious in their meaning:

WinControl The `TWinControl` instance that must handle drops. The windows handle of this control is used in the `RegisterDragDrop` call.

Active Active (accepting drops) or not ? Setting this to `False` will call `RevokeDragDrop`.

FileNamesOnly If this is set to `True`, no `IDropTarget` is used. Instead, the `WM_DROPFILES` message is handled. this is done by calling `DragAcceptFiles` from the ShellApi unit.

OnDropFile If `FileNamesOnly` is `True`, then this event handler is called for each received file.

When the `FileNamesOnly` property is set to `True`, then all that is done is intercepting the `WM_DROPFILES` message. This is done in the `StartDropping` method, where the `WindowProc` message handling method is intercepted:

```
procedure TCustomDropFilesHandler.StartDropping ;
begin
  if Assigned(FControl) then
    begin
    FSavedWndProc:=FControl.WindowProc;
    FControl.WindowProc:=self.ControlWindowProc;
    end;
  FCanDrop:=true;
  TestCanDrop;
end;
```

`TestCandrop` is called on various places and starts the actual drag&drop:

```
procedure TCustomDropFilesHandler.TestCanDrop;

var
  b : Boolean;

begin
  B:=FCanDrop and assigned(FOnDropFile);
  if Assigned(FControl) then
    if FileNamesOnly then
      // we are only interested in the WM_DROPFILES message.
      DragAcceptFiles(FControl.Handle,B)
    else
      begin
      // We want the full DND !
      FDropObject:=TDropObject.Create;
      FDropObject.OnLeave:=Self.OnLeave;
      FDropObject.OnDrop:=Self.OnDrop;
      FDropObject.OnEnter:=Self.OnEnter;
      FDropObject.OnEnter:=Self.OnEnter;
      FDropObject.OnGetDropFormat:=Self.OnGetDropFormat;
      FDropObject.OnDropData:=Self.OnDropData;
```

```
      // This will call RegisterDragDrop !
      FDropObject.SetHandle(FControl.Handle);
      end;
end;
```

The `ControlWindowProc` window handler is very simple, it checks for `WM_DROPFILES`, and if caught, enumerates the dropped filenames. Then the original windows message handler of the control is called.

```
procedure TCustomDropFilesHandler.ControlWindowProc(var aMessage: TMessage);
begin
  if (aMessage.Msg=WM_DROPFILES) and FCanDrop then
    EnumDroppedFiles(Self,aMessage.WParam);
  FSavedWndProc(aMessage);
end;
```

The `EnumDroppedFiles` method is a repeat of what was discussed in the first article:

```
procedure TCustomDropFilesHandler.EnumDroppedFiles(Sender: TObject;
                                                   aWParam: WPARAM);

const
  lMaxFileLength = 1024;

var
  i, lCount: integer;
  aFilename: array [0..lMaxFileLength] of Char;

begin
  if Not assigned(FOnDropFile) then
    Exit;
  try
    // Get file count
    lCount:=DragQueryFile(aWParam, $FFFFFFFF, aFilename, lMaxFileLength);
    // Loop over files
    for i:=0 to lCount-1 do
      begin
      DragQueryFile(aWParam,i,aFilename, lMaxFileLength);
      Try
        // Call event handler.
        FOnDropFile(Sender, aFilename);
      except
        // Do not let the exceptions escape
      end;
      end;
  finally
    DragFinish(aWParam);
  end;
end;
```

When the `Active` property of our `TDropFilesHandler` class is set to `False`, the Drag&Drop must be unregistered, this happens in the `StopDropping` method:

```
procedure TCustomDropFilesHandler.StopDropping;
```

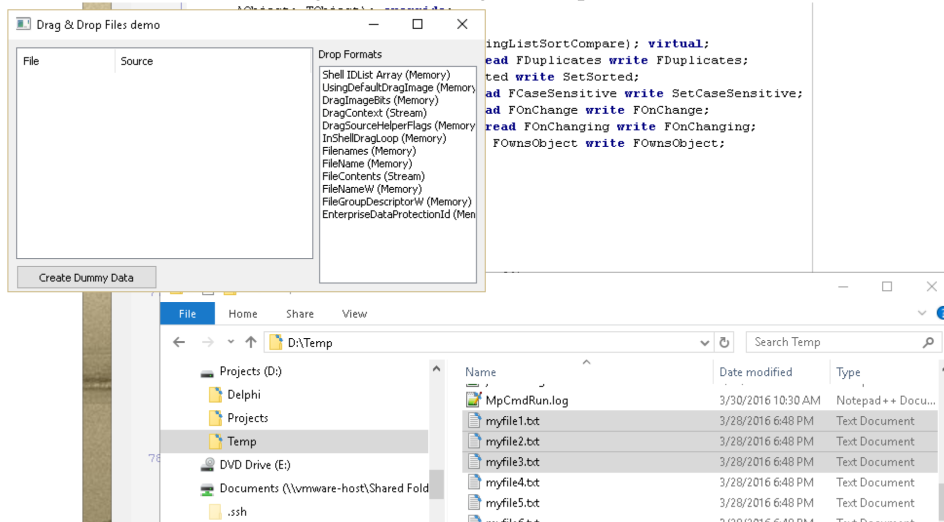


Figure 1: File drag and drop formats

```
begin
  if assigned(FControl) and FControl.HandleAllocated then
    begin
    DragAcceptFiles(FControl.Handle,False);
    if @FSavedWndProc<>nil then
      FControl.WindowProc:=FSavedWndProc;
    end;
  if Assigned(FDropObject) then
    begin
    FDropObject.OnDropData:=Nil;
    FDropObject.OnENter:=Nil;
    FDropObject.OnLeave:=Nil;
    FDropObject.OnOver:=Nil;
    FDropObject.ClearHandle; // Will free
    FDropObject:=Nil;
    end;
  FCanDrop:=False;
end;
```

With that, the interesting code of the component is ready and we can proceed to create some sample programs.

6 File drag and drop

The first example is a simple expansion of the example presented in the previous article. A listview with files in it can now be used to drag files to an application or the windows explorer, but can also accept dropped files. It uses the 2 components introduced here to handle this task. There is a button to generate some dummy data, and the form shows the formats that are dragged to the listview in a listbox, so it can be used to inspect various drag&drop formats. A view of this functionality is presented in figure 1 on page 15

The components are created in code, to avoid having to install the components in the IDE for the demo to work, but these components can of course simply be installed in the IDE on the component palette and dropped from there on the form.

```
procedure TForm1.FormCreate(Sender: TObject);

begin
  // Drag
  FDragFileHandler:=TDragFilesHandler.create(Self);
  FDragFileHandler.OnGetFilesToDrag:=self.DoGetFilesToDrag;
  FDragFileHandler.OnGetStream:=self.DoGetStream;
  FDragFileHandler.WinControl:=ListView1;
  FDragFileHandler.Active:=True;
  // Drop
  FDropFileHandler:=TDropFilesHandler.create(Self);
  FDropFileHandler.OnDrop:=DoDrop;
  FDropFileHandler.OnEnter:=DoDropEnter;
  FDropFileHandler.OnGetDropFormat:=DoGetDropFormat;
  FDropFileHandler.OnDropData:=DoGetDroppedData;
  // In case you are only interested in the names.
  // FDropFileHandler.FilenamesOnly:=true;
  // FDropFileHandler.OnDropFile:=DoDropFile;
  FDropFileHandler.WinControl:=ListView1;
  FDropFileHandler.Active:=True;
end;
```

The drag operation has not essentially changed since the previous article, so we will not focus on it. Note the `DoDropEnter`, `DoGetDropFormat` and `DoGetDroppedData` event handlers: they make up the bulk of the code.

The `DoDropEnter` code is responsible for displaying the available formats. It does this by simply walking the elements in the dynamic array that is passed to it:

```
Procedure TForm1.DoDropEnter(Sender : TObject;
                             Formats : TFormatDescriptionArray;
                             Const X,Y : Integer;
                             Var Effect : Longint) ;


Var
  D : TFormatDescription;
begin
 LBFormats.Items.Clear;
  For D in formats do
  Case D.MediaType of
    TYMED_HGLOBAL :
      LBFormats.Items.Add(D.FormatName + ' (Memory)');
    TYMED_ISTREAM :
      LBFormats.Items.Add(D.FormatName + ' (Stream)');
  End;
  Effect:=DROPEFFECT_COPY;
end;
```

Once a drop occurred, the `DoGetDropFormat` handler will be called several times. To accept files and their contents, we must implement 2 iterations:

1. Get the list of filenames. This can be done by accepting the `CF_NAMES` or `CF_HDROP` formats.
2. Get the file contents. This can be done by reacting to the `CF_CONTENTS` clipboard format.

These iterations are implemented as follows:

```
procedure TForm1.DoGetDropFormat(Sender: TObject;
       Formats: TFormatDescriptionArray;
       const Iteration: Integer;
       var UseFormat, ListCount: Integer);

Var
  I : integer;

begin
  if Iteration=0 then
    begin
    For I:=0 to Length(Formats)-1 do
      if Formats[i].ClipBoardFormat=cf_names then
         UseFormat:=I;
    if UseFormat<0 then
      For I:=0 to Length(Formats)-1 do
        if Formats[i].ClipBoardFormat=cf_hdrop then
           UseFormat:=I;
    end;
  if Iteration=1 then
    begin
    ListCount:=ListView1.Items.Count;
    For I:=0 to Length(Formats)-1 do
      if Formats[i].ClipBoardFormat=cf_contents then
         UseFormat:=I;
    end;
end;
```

After the first iteration, the `DoGetDroppedData` handler will be called. It will receive the filenames, and fill the listview with the names. The second iteration uses the listview's item count to tell the drop component how many times it should fetch clipboard data (the `ListCount` parameter).

Once a clipboard format is selected, the associated data is passed in the `DoGetDroppedData` handler. In the first iteration, it gets the filenames from the stream (2 auxiliary methods from `TDropObject` are used for this:

```
procedure TForm1.DoGetDroppedData(Sender: TObject;
                 Iteration, Format, AListIndex: Integer;
                 var S: TStream);

Var
  L : TStrings;
  lItem :TListItemEx;
  I : Integer;
  D : TDropObject;

begin
  D:=(Sender as TDropObject);
  if Iteration=0 then
    begin
    L:=TStringList.Create;
    try
```

```
    // Get filenames
    Case D.Formats[Format].ClipBoardFormat of
    cf_hdrop :
      D.DropFileNamesToStringList(S,L)
    cf_names :
      D.FileDescriptorToStringList(S,L);
    end;
    // Fill listview.
    ListView1.Items.Clear;
    For I:=0 to L.Count-1 do
      begin
      lItem:=TListItemEx.Create(ListView1.Items);
      ListView1.Items.AddItem(lItem,-1);
      lItem.SubItems.Add(L[i]);
      lItem.Caption:=ExtractFileName(L[i]);
      lItem.Data:=Nil;
      end;
  finally
    L.Free;
  end;
  end;
```

During the second iteration, the file data is passed to this handler. The file data is simply attached to the listview items that were created in the first iteration. In a real-world program, some data structure would probably be set up to receive this data properly.

```
  if Iteration=1 then
    begin
    Litem:=ListView1.Items[AListIndex] as TListItemEx;
    Litem.DataStream:=S;
    S:=Nil;
    end;
end;
```

That is all. The application is now ready to accept files from the explorer or any other program that implements the necessary clipboard formats.

To demonstrate this, the program can be run twice: files can be dropped from the explorer into the first instance, and then a file can be dragged and dropped from the first instance to the second instance. This is shown in figure 2 on page 19

7 Other clipboard formats

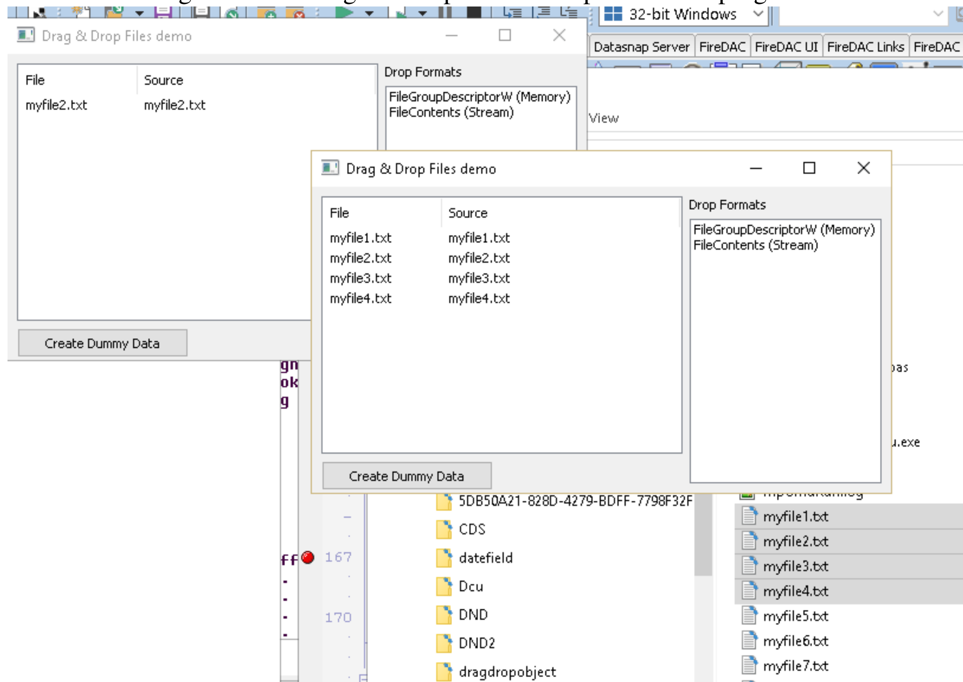
Till now the discussion exclusively focused on drag and drop of filenames. But there is no principal reason for this restriction, because any content can be used in drag and drop: everything depends on the supported formats by both the source and target applications involved in the drag and drop operation. To demonstrate this, we'll make a small application that accepts the HTML clipboard format.

This clipboard format is described on MSDN:

```
https://msdn.microsoft.com/en-us/library/windows/desktop/ms649015%28v=vs.85%29.as
```

The firefox browser implements this format, so it can be used to drag HTML from the

Figure 2: File drag and drop between explorer and 2 programs



browser onto any application that understands it. We'll make a small application with a memo and a listbox, and use this to accept dropped HTML.

The drop handler is set up in the `OnCreate` of the form:

```
procedure TForm2.FormCreate(Sender: TObject);
begin
  // Drop
  FDropFileHandler:=TDropFilesHandler.create(Self);
  FDropFileHandler.WinControl:=Memo1;
  FDropFileHandler.OnEnter:=DoDropEnter;
  FDropFileHandler.OnGetDropFormat:=DoGetDropFormat;
  FDropFileHandler.OnDropData:=DoGetDroppedData;
  FDropFileHandler.Active:=True;
end;
```

Again, the `DoDropEnter` is used to list the available formats. The code is identical to the one in the previous application, it will not be repeated here.

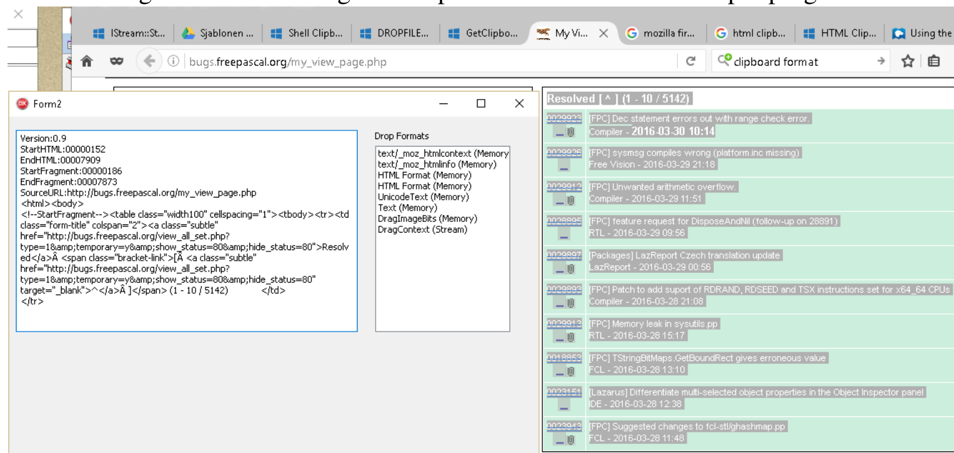
The `DoGetDropFormat` event handler is used to select the HTML clipboard format. It does this by matching the format name:

```
procedure TForm2.DoGetDropFormat(Sender: TObject;
                  Formats: TFormatDescriptionArray;
                  const Iteration: Integer;
                  var UseFormat, ListCount: Integer);

Var
  I : integer;

begin
```

Figure 3: HTML drag and drop between browser and Delphi program




```
  if Iteration=0 then
    For I:=0 to Length(Formats)-1 do
      if Formats[i].FormatName='HTML Format' then
        UseFormat:=I;
end;
```

The `DoGetDroppedData` will be called with the HTML data in the stream:

```
procedure TForm2.DoGetDroppedData(Sender: TObject; Iteration, Format,
  AListIndex: Integer; var S: TStream);

begin
  if Iteration=0 then
    Memo1.Lines.LoadFromStream(S);
end;
```

It doesn't get more simple than this. Obviously, in a real-world application, the HTML data would be analysed and displayed in a control capable of showing HTML. The result of this simple code is shown in figure 3 on page 20

8 Conclusion

In this article, the drag side (source) of DND operations was completed, and the drop side (target) was implemented: there is relatively few code involved in a DnD operation. The principal difficulty lies in understanding the various clipboard formats, and possibly the type of media used in transferring the data: in the code presented here, only the mechanisms involving direct memory copy and IStream interface are explored: these should go a long way in implementing Drag and Drop for the bulk of available applications. For those looking for a more complete implementation of drag and drop, the Drag and Drop component suite handles more types of media, and is capable of converting between types:

`https://github.com/DelphiPraxis/The-Drag-and-Drop-Component-Suite-for-Delphi`

Many thanks go to Peter van der Sman for the initial component implementations and to Anne Zheng for the initial ideas and a critical review of the articles.