How to receive data dragged from other applications (part 5 of 6)

Implementing IDropTarget

Implementing *IDropTarget*

We will illustrate how to implement *IDropTarget* by considering two examples.

In both examples we will implement *IDropTarget* in the main form. Since *TForm* implements *IUnknown* we don't have to bother implementing its methods. Therefore all we need to do is implement the *IDropTarget* methods.

Boilerplate code

Both examples share some boilerplate code, which is presented in *Listing 9*.

```
unit Form1;

interface

type
  TForm1 = class(TForm, IDropTarget)
    ...
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    ...
  private
    ...
  protected
    { IDropTarget methods }
    function IDropTarget.DragEnter = DropTargetDragEnter;
    function DropTargetDragEnter(const dataObj: IDataObject;
      grfKeyState: Longint; pt: TPoint; var dwEffect: Longint): HResult;
      stdcall;
    function IDropTarget.DragOver = DropTargetDragOver;
    function DropTargetDragOver(grfKeyState: Longint; pt: TPoint;
      var dwEffect: Longint): HResult;
      stdcall;
    function IDropTarget.DragLeave = DropTargetDragLeave;
    function DropTargetDragLeave: HResult;
      stdcall;
    function IDropTarget.Drop = DropTargetDrop;
    function DropTargetDrop(const dataObj: IDataObject; grfKeyState:
      Longint; pt: TPoint; var dwEffect: Longint): HResult; stdcall;
  public
    ...
  end;

implementation

function TForm1.DropTargetDragEnter(const dataObj: IDataObject;
  grfKeyState: Integer; pt: TPoint; var dwEffect: Integer): HResult;
begin
  ...
  Result := S_OK;
end;

function TForm1.DropTargetDragLeave: HResult;
begin
  ...
  Result := S_OK;
end;
```

```
function TForm1.DropTargetDragOver(grfKeyState: Integer; pt: TPoint;
  var dwEffect: Integer): HResult;
begin
  ...
  Result := S_OK;
end;

function TForm1.DropTargetDrop(const dataObj: IDataObject;
  grfKeyState: Integer; pt: TPoint; var dwEffect: Integer): HResult;
begin
  ...
  Result := S_OK;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  OleInitialize(nil);
  OleCheck(RegisterDragDrop(Handle, Self));
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  RevokeDragDrop(Handle);
  OleUninitialize;
end;

end.
```

Listing 9

First of all notice that the class declaration for *TForm1* includes *IDropTarget*. Further down in the **protected** section we declare the methods of *IDropTarget*. We have used Delphi's method resolution clauses to rename each of the methods. This has been done because *TForm* already has a method called *DragOver* that clashes with, and gets hidden by, the method of the same name in *IDropTarget*. Each of *IDropTarget*'s methods should return *S_OK* to indicate a successful completion.

The remaining code are the handlers of the form's *OnCreate* and *OnDestroy* events – *FormCreate* and *FormDestroy*. In *FormCreate* we first initialise OLE then call the *RegisterDragDrop* API function to register our implementation of *IDropTarget* – in this case our own form – as the drag-drop handler for the main window. Finally, in *FormDestroy*, we unregister drag-drop for the window by calling *RevokeDragDrop*. Lastly we unitialise OLE.

Registering Drag-drop Implementations

You can register any object that implements *IDropTarget* as the drag-drop handler for any window. It doesn't have to be the main form class or its window.

Example 1: Listing data formats

In this first example we will list all the data formats supported by a data object dropped on a form. Our main window will accept *any* dropped object and will enumerate its data formats when dropped.

To begin, create a new project and add all the boilerplate code from *Listing 9*. Now drop a *TListView* control on the form, name it "lvDisplay", set it's *ViewStyle* property to *vsReport* and add four columns titled "Format", "Storage Medium", "Aspect" and "lindex".

Given that we will accept any object we will always display a copy cursor whenever an object is dragged over the form. Furthermore, we won't take any notice of modifier keys and have no need to examine the object until it is dropped. These observations lead us to implement the *DropTargetDragEnter*, *DropTargetDragLeave* and *DropTargetDragOver* methods as shown in *Listing 10*.

```
function TForm1.DropTargetDragEnter(const dataObj: IDataObject;
  grfKeyState: Integer; pt: TPoint; var dwEffect: Integer): HResult;
begin
  dwEffect := DROPEFFECT_COPY;
  Result := S_OK;
end;

function TForm1.DropTargetDragLeave: HResult;
begin
  Result := S_OK;
```

```
end;

function TForm1.DropTargetDragOver(grfKeyState: Integer; pt: TPoint;
  var dwEffect: Integer): HResult;
begin
  dwEffect := DROPEFFECT_COPY;
  Result := S_OK;
end;
```

Listing 10

All we are doing here is to get Windows to display the copy cursor by setting the *dwEffect* parameter of *DropTargetDragEnter* and *DropTargetDragOver* to *DROPEFFECT_COPY*. *DropTargetDragLeave* is left unchanged from the boilerplate code.

All the action takes place in *DropTargetDrop* as is shown in *Listing 11*.

```
function TForm1.DropTargetDrop(const dataObj: IDataObject;
  grfKeyState: Integer; pt: TPoint; var dwEffect: Integer): HResult;
var
  Enum: IEnumFormatEtc;
  FormatEtc: TFormatEtc;
begin
  OleCheck(DataObj.EnumFormatEtc(DATADIR_GET, Enum));
  lvDisplay.Clear;
  while Enum.Next(1, FormatEtc, nil) = S_OK do
    DisplayDataInfo(FormatEtc);
  dwEffect := DROPEFFECT_COPY;
  Result := S_OK;
end;

procedure TForm1.DisplayDataInfo(const FmtEtc: TFormatEtc);
var
  LI: TListItem;
begin
  LI := lvDisplay.Items.Add;
  LI.Caption := CBFormatDesc(FmtEtc.cfFormat);
  LI.SubItems.Add(TymedDesc(FmtEtc.tymed));
  LI.SubItems.Add(AspectDesc(FmtEtc.dwAspect));
  LI.SubItems.Add(IntToStr(FmtEtc.lindex));
end;
```

Listing 11

First we set the cursor to *DROPEFFECT_COPY* and clear the list view. Next we get a data format enumerator from the data object. Using the enumerator we get each supported data format and display information from it's *TFormatEtc* structure using a subsidiary method, *DisplayDataInfo*. Finally we return *S_OK*.

DisplayDataInfo simply adds a new item to the list view that displays information about the *FmtEtc* structure passed to the method as a parameter. The following helper routines are used to get the display information:

- *CBFormatDesc* – returns a description of the clipboard format specified by *FmtEtc.cfFormat*. The function can handle both built-in and custom clipboard formats.
- *TymedDesc* – returns the name of the *TYMED_** constant that describes the value of *FmtEtc.tymed*.
- *AspectDesc* – returns the name of the *DVASPECT_** constant describing *FmtEtc.dwAspect*.

These routines are not listed here because they are trivial and not relevant to the main subject. However the routines are supplied with the associated *demo program*.

Example 2: Displaying data from selected data formats

This example is more like somethings you may need to implement in a real world application – it checks the data object being dragged and decides whether it can accept the object or not. If it can accept the object the program displays a textual representation of the data.

Our application will accept the following data object types, all of which must be provided via a global memory storage medium:

- Plain text in the *CF_TEXT* format. Text dropped on the application will be displayed in the main window.
- HTML code in the custom "HTML format". The HTML source code will be displayed in full in the main window. The plain text version of the code will also be displayed if it is available.

The program will also attempt to move the data from the source application if the SHIFT key is held down. Otherwise if the either no or any other modifier key is held down the data will be copied.

To begin with, start a new application and drop two *TMemo* controls onto the main form. Arrange them one above the other and name the top one "edText" and the lower one "edHTML". Now add the boilerplate code from *Listing 9*.

Before we get started on the code proper, recall that we will be handling "HTML Format" data objects. Since this is a custom format we need to register it. We do this by declaring a global variable named *CF_HTML* in the form unit's **implementation** section and by registering the format with Windows in the **initialization** section, storing the format identifier in *CF_TEXT*. *Listing 12* illustrates the code.

```
...

implementation

...

var  CF_HTML: TClipFormat; // identifier for HTML clipboard format

...

initialization

CF_HTML := RegisterClipboardFormat('HTML Format');

end.
```

Listing 12

We will begin by considering the *IDropTarget.DragEnter* method, implemented as *DropTargetDragEnter*, as shown in *Listing 13* below. The listing also shows two helper methods called by *DropTargetDragEnter*.

```
function TForm1.DropTargetDragEnter(const dataObj: IDataObject;
  grfKeyState: Integer; pt: TPoint; var dwEffect: Integer): HResult;
begin
  Result := S_OK;
  fCanDrop := CanDrop(dataObj);
  dwEffect := CursorEffect(dwEffect, grfKeyState);
end;

function TForm1.CanDrop(const DataObj: IDataObject): Boolean;
begin
  Result := DataObj.QueryGetData(MakeFormatEtc(CF_TEXT)) = S_OK;
  if not Result then
    Result := DataObj.QueryGetData(MakeFormatEtc(CF_HTML)) = S_OK;
end;

function TForm1.CursorEffect(const AllowedEffects: Longint;
  const KeyState: Integer): Longint;
begin
  Result := DROPEFFECT_NONE;
  if fCanDrop then
  begin
    if (KeyState and MK_SHIFT = MK_SHIFT) and
      (DROPEFFECT_MOVE and AllowedEffects = DROPEFFECT_MOVE) then
      Result := DROPEFFECT_MOVE
    else if (DROPEFFECT_COPY and AllowedEffects = DROPEFFECT_COPY) then
      Result := DROPEFFECT_COPY;
  end;
end;
```

Listing 13

After setting the required return value in *DropTargetDragEnter*, the first thing we do is check if the data object can be dropped, i.e. if it supports either of the data formats we are interested in. We hand this decision

off to *CanDrop* and store the result in the private *fCanDrop* field for use later (in *DropTargetDragOver*).

CanDrop simply queries the data object to see if it supports the *CF_TEXT* data format. If it doesn't the data object is queried again for the HTML format. If neither format is supported false is returned. Note that *CanDrop* calls the convenience *MakeFormatEtc* method that simply constructs an appropriate *TFormatEtc* structure to pass to *IDataObject.QueryGetData*.

Back in *DropTargetDragEnter* the next thing we do is to determine the required cursor effect. Again this is handed off to a helper method, *CursorEffect*. *DropTargetDragEnter* passes the bitmask of allowed cursor effects, from its *dwEffect* parameter, along with the keyboard and mouse state, from the *grfKeyState* parameter, to *CursorEffect*. This method calculates the required cursor effect according to the following rules:

- If the data object does not support the required data formats return *DROPEFFECT_NONE*.
- If the SHIFT key is pressed and the *DROPEFFECT_MOVE* effect is allowed return *DROPEFFECT_MOVE*.
- If any other, or no, modifier keys are pressed, or if *DROPEFFECT_MOVE* is reqested but not allowed, *DROPEFFECT_COPY* is returned.

Listing 14 shows the next *IDropTarget* method, *DragOver*, implemented as *DropTargetDragOver*.

```
function TForm1.DropTargetDragOver(grfKeyState: Integer; pt: TPoint;
  var dwEffect: Integer): HResult;
begin
  Result := S_OK;
  dwEffect := CursorEffect(dwEffect, grfKeyState);
end;
```

Listing 14

All we do here is set the cursor effect once again, using the current key state from the *grfKeyState* parameter and the permitted cursor effects from *dwEffect*. The cursor state may change between calls to this method because the pressed modifier keys may have changed.

Next up is *DragLeave*, implemented as *DropTargetDragLeave*. As is shown by *Listing 15*, this method is unchanged from the *boilerplate code*.

```
function TForm1.DropTargetDragLeave: HResult;
begin
  Result := S_OK;
end;
```

Listing 15

The final *IDropTarget* method to consider is *Drop*, implemented as *DropTargetDrop*. *Listing 16* shows this method along with helper methods that are used to display the data.

```
function TForm1.DropTargetDrop(const dataObj: IDataObject;
  grfKeyState: Integer; pt: TPoint; var dwEffect: Integer): HResult;
begin
  Result := S_OK;
  fCanDrop := CanDrop(dataObj);
  dwEffect := CursorEffect(dwEffect, grfKeyState);
  DisplayData(dataObj);
end;

procedure TForm1.DisplayData(const DataObj: IDataObject);
begin
  edText.Text := GetTextFromObj(DataObj, CF_TEXT);
  edHTML.Text := GetTextFromObj(DataObj, CF_HTML);
end;

function TForm1.GetTextFromObj(const DataObj: IDataObject;
  const Fmt: TClipFormat): string;
var
  Medium: TStgMedium;
  PText: PChar;
begin
  if DataObj.GetData(MakeFormatEtc(Fmt), Medium) = S_OK then
  begin
    Assert(Medium.tymed = MakeFormatEtc(Fmt).tymed);
```

```
    try
      PText := GlobalLock(Medium.hGlobal);
      try
        Result := PText;
      finally
        GlobalUnlock(Medium.hGlobal);
      end;
    finally
      ReleaseStgMedium(Medium);
    end;
  end
  else
    Result := '';
end;
```

Listing 16

DropTargetDrop starts by re-checking data object to see if we can handle it and then sets the drop cursor. Finally it calls *DisplayData* to display the data from the dropped data object.

DisplayData simply sets the text of the two memo controls to the strings returned from *GetTextFromObj*. *GetTextFromObj* is called twice, once to retrieve any plain text (*CF_TEXT*) and again to retrieve any HTML Format (*CF_HTML*) data as text from the data object. *GetTextFromObj* will return the empty string if the requested data format is not available.

GetTextFromObj extracts text data from the data object in the format specified in its *Fmt* parameter. It does this by calling *MakeFormatEtc* to create a *TFormatEtc* structure that describes the desired clipboard format. This structure is passed to the data object's *GetData* method. If the method returns a value other than *S_OK* then the requested format is not supported and the empty string is returned. Otherwise the now familiar code to retrieve text from a global memory handle is executed and the text is returned.

All that remains to do now is to implement the *MakeFormatEtc* helper method referred to above. The method is presented in *Listing 17* below.

```
function TForm1.MakeFormatEtc(const Fmt: TClipFormat): TFormatEtc;
begin
  Result.cfFormat := Fmt;
  Result.ptd := nil;
  Result.dwAspect := DVASPECT_CONTENT;
  Result.lindex := -1;
  Result.tymed := TYMED_HGLOBAL;
end;
```

Listing 17

All the method does is set up and return a *TFormatEtc* structure that requires a *TYMED_HGLOBAL* medium type for the clipboard format passed in the method's *Fmt* parameter. The code should be familiar by now.

Our examination of how to catch data dragged and dropped from other applications is now complete. In the *last section* we will wrap up the article and provide a link to the article's source code.
