Dragging And Dropping Part 3: Windows

by Brian Long

Over the last two months we have investigated VCL support for in-application drag and drop. This month we branch out and look at inter-application drag and drop. This focuses on how to make a Delphi application act as a recipient for information dragged from other Windows applications.

It is also possible to act as the source of a drag operation, so your users can drag information to other applications, but space restricts us to looking at initiating drag operations this time.

The Old 16-Bit Windows Way

Windows 3.x supported a mechanism that existed to allow users to select one or more files in File Manager, drag them over your application and drop them on one of your application windows. This mechanism is also supported in 32-bit Windows in the same way, allowing you to accept files dragged from Windows Explorer.

To tell Windows that one of your windows should be treated as a drop target, you pass its window handle to the `DragAcceptFiles` API. This routine also takes a `Boolean` acceptance parameter. You pass in `True` to specify your window will accept files dropped onto it, or `False` to discontinue accepting dropped files before terminating.

wm_DropFiles

Assuming you have told Windows that you have a window that will accept dropped files, then when the user does drop one or more files onto your window (or one of its children) a `wm_DropFiles` message is sent to it. If an application processes this message it should set the `Result` field to `0` to inform Windows not to do its own default processing.

The `WParam` value sent with this message contains the only useful information (`LParam` is undefined), which is a memory handle to a drop structure. The Microsoft documentation typically describes this as an internal drop structure, since in 16-bit Windows it was undocumented. However, the Win32 SDK provides us with the definition of the `DROPFILES` structure, which can be referred to as `TDropFiles` in Delphi.

The record, as defined in the `ShlObj` unit and shown in Listing 1, holds information on the location of the cursor when the files were dropped as well as the list of filenames that were dropped.

Given this information, you could use standard Windows memory management code to access this record in the message handler. However, instead of analysing the data ourselves, we are advised to use a number of dedicated APIs. This gets over the lack of definition of the record in 16-bit Windows and also the issue of catering for Unicode characters, should they come up.

`DragQueryPoint` takes a memory handle to a dropped file structure as well as a pass by reference `TPoint` parameter. The point record is filled with the drop point information from the dropped file structure.

`DragQueryFile` is designed to return information about one of the dropped files. It takes a file index number (which file to return information about) along with a character buffer and buffer size and fills the buffer with the file name and path. If no buffer is supplied, the function returns the size of the required buffer. A file index of `-1` causes the function to return the number of dropped files.

Finally, `DragFinish` frees the memory occupied by a dropped file structure.

With this information, you can write code that lists all dropped files in a listbox, as shown in Listing 2. This code can be found in the OldDrag.Dpr project on the disk.

➤ *Listing 1: TDropFiles record.*

```
PDropFiles = ^TDropFiles;
_DROPFILES = record
  pFiles: DWORD;        { offset of file list }
  pt: TPoint;           { drop point (client coords) }
  fNC: BOOL;            { is it on NonClient area and pt is in screen coords }
  fWide: BOOL;          { WIDE character switch }
end;
TDropFiles = _DROPFILES;
DROPFILES = _DROPFILES;
```

➤ *Listing 2: Accessing dropped file information.*

```
procedure TForm1.WMDropFiles(var Msg: TWMDropFiles);
var
  Pt: TPoint;
  Count, Loop: Integer;
  Buf: array[0..MAX_PATH] of Char;
begin
  try
    Msg.Result := 0;
    DragQueryPoint(Msg.Drop, Pt);
    Caption := Format('Files dropped at (%d,%d)', [Pt.X, Pt.Y]);
    Count := DragQueryFile(Msg.Drop, Cardinal(-1), Buf, SizeOf(Buf));
    for Loop := 0 to Pred(Count) do begin
      DragQueryFile(Msg.Drop, Loop, Buf, SizeOf(Buf));
      lstFiles.Items.Add(StrPas(Buf))
    end
  finally
    DragFinish(Msg.Drop)
  end
end;
```

This project works in all Windows versions of Delphi.

The Newer, Win32 COM Way

Windows 95 introduced a more open (and involved) mechanism for inter-application drag and drop, which allows you to drag information in numerous formats (simultaneously). It is based around COM and involves part of the OLE Windows subsystem.

To operate successfully you must initialise OLE before calling any of the routines and uninitialise it afterwards, with calls to `OleInitialize` and `OleUninitialize` respectively. Also, an application calls `RegisterDragDrop` to register a window as a drop target. Then, when done, `RevokeDragDrop` stops it being a drop target.

IDropTarget

When registering a window as a drop target you pass in the window handle and a reference to an `IDropTarget` interface whose methods will be called to control the drag/drop operation. `IDropTarget` can be implemented in any object, not necessarily the one that is registered as a drop target window (see the later comments about Delphi 3 forms and COM). You can see `IDropTarget` in Listing 3.

When the mouse is dragged into a registered drop target window `IDropTarget.DragEnter` is called. This allows the drag operation to be accepted or cancelled and allows the cursor to be customised to give user feedback. The `dataObj` parameter is another interface reference, representing a data object that can describe and render (if need be) the data being dragged. The `grfKeyState` parameter gives information on the standard shift keys that are held down (much like the `Shift` parameter in `OnKeyDown/Up` and `OnMouseDown/Move/Up` event handlers). `Pt` specifies the mouse cursor location, whilst `dwEffect` is a `var` parameter that can be used to cancel the drag or specify the effect of the drag (link, move or copy the data).

As the mouse is moved around the window, `IDropTarget.DragOver` is repeatedly called to provide user feedback and to allow the drag operation to potentially be customised further. For example, if the user changes which of the shift keys are pressed, the effect of the drag can be modified.

If the mouse is moved out of the window, or the drag operation is cancelled, `IDropTarget.DragLeave` is called. Here you must remove whatever user feedback you have set up and drop any references to the data object passed in `IDropTarget.DragEnter`.

Finally, if the item is dropped in the window (the mouse cursor is released) `IDropTarget.Drop` is called. Here you incorporate the dragged source data into the target window however is appropriate and then do the same tidying up as in `IDropTarget.DragLeave`.

```
IDropTarget = interface(IUnknown)
  ['{00000122-0000-0000-C000-000000000046}']
  function DragEnter(const dataObj: IDataObject; grfKeyState: Longint;
    pt: TPoint; var dwEffect: Longint): HResult; stdcall;
  function DragOver(grfKeyState: Longint; pt: TPoint; var dwEffect: Longint):
    HResult; stdcall;
  function DragLeave: HResult; stdcall;
  function Drop(const dataObj: IDataObject; grfKeyState: Longint; pt: TPoint;
    var dwEffect: Longint): HResult; stdcall;
end;
```

➤ *Listing 3: The IDropTarget interface.*

```
IDataObject = interface(IUnknown)
  ['{0000010E-0000-0000-C000-000000000046}']
  function GetData(const formatetcIn: TFormatEtc; out medium: TStgMedium):
    HResult; stdcall;
  function GetDataHere(const formatetc: TFormatEtc; out medium: TStgMedium):
    HResult; stdcall;
  function QueryGetData(const formatetc: TFormatEtc): HResult; stdcall;
  function GetCanonicalFormatEtc(const formatetc: TFormatEtc; out formatetcOut:
    TFormatEtc): HResult; stdcall;
  function SetData(const formatetc: TFormatEtc; var medium: TStgMedium;
    fRelease: BOOL): HResult; stdcall;
  function EnumFormatEtc(dwDirection: Longint; out enumFormatEtc:
    IEnumFormatEtc): HResult; stdcall;
  function DAdvise(const formatetc: TFormatEtc; advf: Longint; const advSink:
    IAdviseSink; out dwConnection: Longint): HResult; stdcall;
  function DUnadvise(dwConnection: Longint): HResult; stdcall;
  function EnumDAdvise(out enumAdvise: IEnumStatData): HResult;
    stdcall;
end;
```

➤ *Listing 4: The IDataObject interface.*

IDataObject

The data object (as represented by the `IDataObject` parameter in the `DragEnter` and `Drop` methods of `IDropTarget`) warrants some investigation now, as it is the mechanism by which we initially decide if we will accept the drop and also how we get the dragged information if it is dropped.

The primary job of the `IDataObject` interface (see Listing 4) is to transfer data from a source (the window that was dragged from) to a target (the window that was dropped on). The data can be available in numerous formats, each format being stored in its own storage medium. Optionally, the data might be rendered for a specific target device.

The data object allows the drop target to query it to see if a requested data form is available in a specified storage medium. The drop target can also enumerate all the supported formats, and can be notified of changes in the data by setting up an advisory sink (the drop target application can implement the `IAdviseSink` interface, called when things change).

Clipboard Formats

As I explored the subject of data objects, I was surprised to find that there was a lot in common between OLE drag and drop and the clipboard. In particular, a data object can be placed on the clipboard thereby making all the data formats managed by the data object available from the clipboard.

If you think about it, this makes a certain amount of sense, because you can drag potentially complex data between applications in one motion, or alternatively copy it

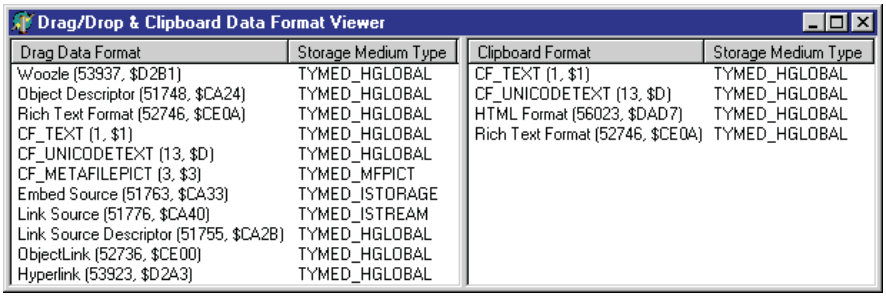
into the clipboard and then paste it into another application using either the `Edit | Paste` or `Edit Paste Special...` menu items. The `Paste Special` dialog lists all the supported data formats that are in the clipboard.

To get an idea of what formats are used by some of the more common applications, I have written a program to help (on the disk as ClipFmtList.dpr). This application knows how to query all the formats maintained by a data object. It has two listviews on it.

The right-hand listview is regularly populated by a timer and shows what data formats are available in the clipboard at any given time. The code asks Windows to return a reference to a data object that contains the clipboard data in all its formats. Even if no application has placed a data object in the clipboard, Windows can manufacture one if asked.

The left-hand listview is registered as a drop target and lists all the formats available when information is dragged into it. It gains access to the drag object in charge of the operation's data and iterates through it.

Figure 1 shows the program after some HTML has been copied into the clipboard from within Internet Explorer and a block of text from Word has been dragged across the left listview. You can see a few



➤ *Figure 1: A display of clipboard formats available through drag and drop, and through the clipboard.*

common formats used by both Internet Explorer and Word, such as `CF_TEXT`, `CF_UNICODETEXT` and `Rich Text Format`. You might also see various other clipboard formats, unique to the application manipulating the clipboard or data object. For example, Word has a data format called `Woozle`.

The formats listed as identifiers with a `CF_` prefix are standard Windows clipboard formats which are always available. Other formats are custom clipboard formats that are initialised by various applications. Any application can use any of these formats by passing the exact same clipboard format string (of which `Rich Text Format` is an example) along to the `RegisterClipboardFormat` function.

The function returns a numeric value which uniquely identifies that clipboard format within the current Windows session. The numbers seen after the clipboard formats in Figure 1 are these clipboard format identifiers. You can see that the standard formats all have very low values, whereas custom formats seem to use values over 50,000. Next to each clipboard format is information about which medium type the data is stored in. Typical values indicate global memory handles, `IStorage` and `IStream` interfaces, and a memory handle to a `TMetaFilePict` record.

The program's implementation is straightforward. Each timer tick the program asks the clipboard for a data object that can manage the data therein. This object is passed to a utility routine called `ListFormats` which takes a data object and also a `TListItems` object (the `Items` property of a listview). `ListFormats` then uses the data object's `EnumFormatEtc` method to iterate across all data formats and add their details to the listview.

Each supported format is represented by a `TFormatEtc` record (in the Win32 SDK help this is listed as the original C `FORMATETC` structure) which contains the clipboard format (a `TClipFormat` field called `cfFormat`) and the storage medium type (an integer called `tymed`).

Other fields allow a format to be targeted at a specific device, and specify how much detail should be used, but these do not seem to be used very often. `ListFormats` uses some helper routines to take a clipboard format and turn into the representative string, and do the same for a storage medium type value. Listing 5 shows the code discussed so far.

The other part of this utility shows what clipboard formats are available through a COM drag and

➤ *Listing 5: Listing all available clipboard formats via a data object.*

```
procedure TDataFormatListForm.ListFormats(List: TListItems;
  DataObj: IDataObject);
var
  EFE: IEnumFormatEtc; //enumeration interface
  FE: TFormatEtc; //Clipboard format, storage medium type etc.
  CElt: Longint; //count of elements returned
  Item: TListItem;
begin
  OleCheck(dataObj.EnumFormatEtc(DATADIR_GET, EFE));
  List.BeginUpdate;
  try
    List.Clear;
    CElt := -1;
    while CElt <> 0 do begin
      OleCheck(EFE.Next(1, FE, @CElt));
      if CElt > 0 then begin
        Item := List.Add;
        Item.Caption := ClipFormatToStr(FE.cfFormat);
        Item.SubItems.Add(TyMedToStr(FE.tymed));
      end
    end
  finally
    List.EndUpdate
  end;
end;
procedure TDataFormatListForm.TimerTimer(Sender: TObject);
var DataObj: IDataObject;
begin
  if Succeeded(OleGetClipboard(DataObj)) then
    ListFormats(lstClipFmt.Items, DataObj)
end;
```

```
TDataFormatListForm = class(TForm, IUnknown, IDropTarget)
  lstDragFmt: TListView;
...
private
  //IDropTarget
  function DragEnter(const dataObj: IDataObject;
    grfKeyState: Longint; pt: TPoint;
    var dwEffect: Longint): HResult; stdcall;
  function DragOver(grfKeyState: Longint; pt: TPoint;
    var dwEffect: Longint): HResult; reintroduce; stdcall;
  function DragLeave: HResult; stdcall;
  function Drop(const dataObj: IDataObject;
    grfKeyState: Longint; pt: TPoint;
    var dwEffect: Longint): HResult; stdcall;
  ...
end;
...
function TDataFormatListForm.DragEnter(
  const dataObj: IDataObject; grfKeyState: Integer;
  pt: TPoint; var dwEffect: Integer): HResult;
begin
  ListFormats(lstDragFmt.Items, dataObj);
  Result := S_OK
end;
function TDataFormatListForm.DragLeave: HResult;
begin
  Result := S_OK
end;
function TDataFormatListForm.DragOver(grfKeyState: Integer;
  pt: TPoint; var dwEffect: Integer): HResult;
begin
  Result := S_OK
end;
function TDataFormatListForm.Drop(const dataObj:
  IDataObject; grfKeyState: Integer; pt: TPoint;
  var dwEffect: Integer): HResult;
begin
  DragLeave; //Call routine that potentially does tidying up
  Result := S_OK
end;
procedure TDataFormatListForm.FormCreate(Sender: TObject);
begin
  OleCheck(RegisterDragDrop(lstDragFmt.Handle, Self));
  //Make sure timer ticks immediately
  if Assigned(Timer.OnTimer) then
    Timer.OnTimer(Timer)
end;
procedure TDataFormatListForm.FormDestroy(Sender: TObject);
begin
  OleCheck(RevokeDragDrop(lstDragFmt.Handle))
end;

...
initialization
  OleCheck(OleInitialize(nil))
finalization
  OleUninitialize
end.
```

➤ *Listing 6: Implementing IDropTarget.*

drop operation. This requires implementing the four methods of `IDropTarget`. This is simple in the case of `DragOver` and `DragLeave`, which need do nothing in this case apart from return a success value.

`Drop` does much the same, although it calls `DragLeave` to take advantage of any tidying up that *could* be done there. All this leaves is `DragEnter`, which passes the drag object parameter to `ListFormats` again to get all the information on the screen.

Delphi 3 Issues

Listing 6 shows the gory details. Notice the drag/drop registration and unregistration being performed in the `OnCreate` and `OnDestroy` event handlers respectively. This code works well in Delphi 4 and later but causes a problem with Delphi 3. When implementing the `IDropTarget` interface in the form, you might notice that I only implemented the `IDropTarget` methods. I did not implement the methods of `IUnknown`, the interface that `IDropTarget` is based on.

This is because `TForm` inherits from `TComponent`, which implements the `IUnknown` methods (along with `IDispatch` methods). It implements the methods, but the definition of `TComponent` does not claim to implement `IUnknown` (by having `IUnknown` in the brackets in the first line of the class definition). When you implement an interface in a form and specify that the form supports your interface and `IUnknown`, the already existing `IUnknown` method implementations will be automatically used.

```
function TComponent.QueryInterface(const IID: TGUID; out Obj): HResult;
begin
  if FVCLComObject = nil then begin
    if GetInterface(IID, Obj) then
      Result := S_OK
    else
      Result := E_NOINTERFACE
  end else
    Result := IVCLComObject(FVCLComObject).QueryInterface(IID, Obj);
end;
function TComponent._AddRef: Integer;
begin
  if FVCLComObject = nil then
    Result := -1   // -1 means no ref. counting is taking place
  else
    Result := IVCLComObject(FVCLComObject)._AddRef;
end;
function TComponent._Release: Integer;
begin
  if FVCLComObject = nil then
    Result := -1   // -1 means no ref. counting is taking place
  else
    Result := IVCLComObject(FVCLComObject)._Release;
end;
```

➤ *Listing 7: Delphi 4's IUnknown methods in TComponent.*

The problem arises because of what these methods do. In Delphi 4 and later, they check whether the `VCLComObject` property has been assigned a value (by examining the `FVCLComObject` data field). As Listing 7 shows, if no COM object interface has been assigned to this property, no reference counting is performed, otherwise, the COM object's reference counting methods are employed.

Listing 8 shows that Delphi 3 code assumes that `VCLComObject` will have been assigned. If it hasn't been assigned (our code does not assign it a value), or the methods have not been re-implemented, the code generates an Access Violation. The code on the disk includes implementations of the `IUnknown` methods for Delphi 3 users to circumvent the failure.

Another solution would have been to implement `IDropTarget` in another object. This would have avoided Windows trying to talk to the form using COM, triggering the problem in the first place. Alternatively you could implement `IUnknown` in another object and assign the object's `IUnknown` interface reference to the `VCLComObject` property (after typecasting it to a pointer).

`VCLComObject` is used to allow a component to support COM, albeit

by delegating `IUnknown` and `IDispatch` calls to another object.

A Better Example

Having spent some time looking at a simple application that interacts with the COM drag/drop system, let's now look at a more involved example. The COMDragDrop.dpr project works with Delphi 3 and later and acts as a fairly generic drop target.

With the program running, you can drag data from any suitable application onto it and it displays the data in as many ways as possible, taking into account all the available clipboard formats. This is done using a page control, with one page per supported format. Only the pages for available formats become visible when data is dropped onto the form. Figure 2 shows the result.

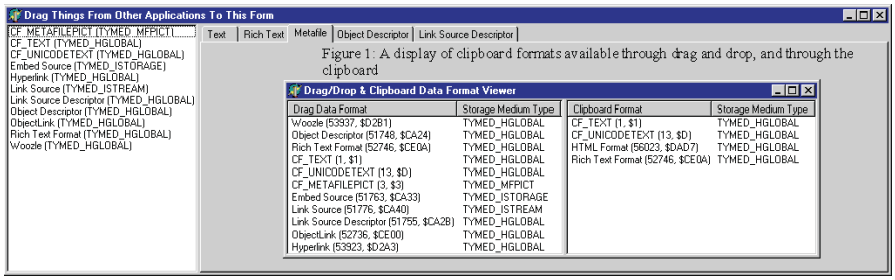
The architecture of the program is like this. A class called `TDataObject` is designed to provide easy access to the data in a data object. An `IDataObject` interface reference is passed to the `TDataObject` constructor to start it off.

`TDataObject` serves three purposes. Firstly, it has a `ListFormats` method which enumerates all the supported data formats and populates a listview with them. The implemention is similar to that in the ClipFmtList.dpr project.

Secondly, the constructor initialises a set property called `DataFormats`. This allows the rest of the program to easily identify which formats are available in the data object. An enumerated type is used to define the values that can go in the set property. These values are easier to use than the original clipboard formats. To cover as many data formats as possible, the code registers some of the non-standard, but common, clipboard formats, such as `Rich Text Format`.

Rather than enumerating the data formats again, `DataFormats` is set up by querying the data object for each format known to the object. Any that are indicated as

➤ *Figure 2: The drag and drop application in action.*



➤ *Listing 8: Delphi 3's IUnknown methods in TComponent.*

```
function TComponent.QueryInterface(const IID: TGUID; out Obj): Integer;
begin
  Result := IVCLComObject(FVCLComObject).QueryInterface(IID, Obj);
end;
function TComponent._AddRef: Integer;
begin
  Result := IVCLComObject(FVCLComObject)._AddRef;
end;
function TComponent._Release: Integer;
begin
  Result := IVCLComObject(FVCLComObject)._Release;
end;
```

➤ *Listing 9: The IDataObject wrapper class.*

```
type
  //When adding to this set, update the GetDataFormats
  //function as well
  TDataFormat = (dfText, dfHDrop, dfDIB, dfBitmap,
    dfPalette, dfWMF, dfEMF, dfRTF, dfFileName,
    dfShellIDList, dfObjectDescriptor, dfLinkSrcDescriptor);
  TDataFormats = set of TDataFormat;
  TDataObject = class
  private
    FDataObject: IDataObject;
    FFormatEtc: TFormatEtc;
    FDataFormats: TDataFormats;
    //Stores data object's data formats in FDataFormats
    procedure GetDataFormats;
    procedure SetupFormatEtc(ClipFmt: TClipFormat;
      TyMed: Longint);
    procedure GetDescriptor(SM: TStgMedium; List: TStrings);
  public
    constructor Create(DataObj: IDataObject);
    procedure GetDataAsBitmap(Bmp: TBitmap);
    procedure GetDataAsDIB(Bmp: TBitmap);
    procedure GetDataAsHDrop(FileList: TStrings);
    procedure GetDataAsWMF(MetaFile: TMetaFile);
    procedure GetDataAsEMF(MetaFile: TMetaFile);
    procedure GetDataAsPalette(Bmp: TBitmap);
    procedure GetDataAsRTF(var Txt: String);
    procedure GetDataAsText(var Txt: String);
    procedure GetDataAsFileName(var Txt: String);
    procedure GetDataAsShellIDList(IDList: TStrings);
    procedure GetDataAsObjectDescriptor(ObjDescList:
      TStrings);
    procedure GetDataAsLinkSrcDescriptor(
      LinkSrcDescList: TStrings);
    procedure ListFormats(List: TStrings);
    property DataFormats: TDataFormats read FDataFormats;
  end; //TDataObject
constructor TDataObject.Create(DataObj: IDataObject);
begin
  inherited Create;
  FDataObject := DataObj;
  GetDataFormats
end;
procedure TDataObject.SetupFormatEtc(ClipFmt: TClipFormat;
  TyMed: Longint);
begin
  FFormatEtc.cfFormat := ClipFmt;
  FFormatEtc.tymed := TyMed;
  FFormatEtc.ptd := nil;
  FFormatEtc.dwAspect := DVASPECT_CONTENT;
  FFormatEtc.lindex := -1;
end;
procedure TDataObject.GetDataFormats;
  procedure GetDataFormat(ClipFmt: TClipFormat;
    TyMed: Longint; Format: TDataFormat);
  begin
    SetupFormatEtc(ClipFmt, TyMed);
    if FDataObject.QueryGetData(FFormatEtc) = S_OK then
      Include(FDataFormats, Format);
  end;
begin
  FDataFormats := [];
  GetDataFormat(CF_BITMAP,  TYMED_GDI,     dfBitmap);
  GetDataFormat(CF_DIB,     TYMED_HGLOBAL, dfDIB);
  GetDataFormat(CF_HDROP,   TYMED_HGLOBAL, dfHDrop);
  GetDataFormat(CF_METAFILEPICT, TYMED_MFPICT,  dfWMF);
  GetDataFormat(CF_ENHMETAFILE,  TYMED_ENHMF,   dfEMF);
  GetDataFormat(CF_PALETTE,  TYMED_GDI,     dfPalette);
  GetDataFormat(CF_TEXT,     TYMED_HGLOBAL, dfText);
  GetDataFormat(CF_RTF,      TYMED_HGLOBAL, dfRTF);
  GetDataFormat(CF_FILENAME, TYMED_HGLOBAL, dfFileName);
  GetDataFormat(CF_IDLIST,   TYMED_HGLOBAL, dfShellIDList);
  GetDataFormat(CF_OBJECTDESCRIPTOR,  TYMED_HGLOBAL,
    dfObjectDescriptor);
  GetDataFormat(CF_LINKSRCDESCRIPTOR, TYMED_HGLOBAL,
    dfLinkSrcDescriptor);
end;
```

being supported are added into the `DataFormats` property. You can see this happening in Listing 9, along with one of the helper routines, called `SetupFormatEtc`.

The final job for this object is to provide simple ways to get the data out of the data object, without having to resort to API-level work every time you need access to the data. That is the reasoning behind the various `GetDataAsXXXX` methods declared in Listing 9.

Accessing The Data

Notice that all these methods are procedures, none are functions. Whether data is returned through a parameter or a function result is fairly irrelevant in the case of strings and integers etc. However, things change when the information being returned is an object.

If a function returns, say, a `TBitmap` object, the function will have typically created the object in order to return it. This places an immediate responsibility on the caller to remember to destroy it. If the calling code calls the function and neglects to destroy the bitmap object, you have an instant resource leak.

It is typically better to define a `TBitmap` parameter to the method. The caller is then responsible for obtaining the bitmap (possibly by creating it) and destroying it if necessary. A paired responsibility of creating and destroying is much better than a responsibility for just destroying an object.

Each of these methods does much the same job. It first sets up a `TFormatEtc` record for the right clipboard format and storage medium that is expected. This record is passed to the data object's `GetData` method. Assuming all went well, the data object's `GetData` method will fill up a `TStgMedium` record (documented in the Win32 SDK help as a `STGMEDIUM` structure).

`GetData` will have allocated space for a copy of its data to be rendered into the specified storage medium. The `TStgMedium` variant record holds a reference to the allocated storage medium. As Listing 10 shows, this is either as a bitmap handle, a global memory handle for the data block, a global memory handle for a `TMetaFilePict` record, a handle to an enhanced Windows metafile, a wide character file name, or an `IStream` or `IStorage` interface reference. The appropriate field to use is dictated by the storage medium type in the `TFormatEtc` record as shown in Table 1.

Once you have accessed the data in the format that you understand, there is a requirement to free the resources used to keep hold of the copy of the dragged data. This is easily done by passing the `TStgMedium` record to the `ReleaseStgMedium` routine: this examines the passed record and takes appropriate steps to free the memory occupied by the data.

Applications that act as drag sources vary. Sometimes the drop target is meant to free the data space, sometimes the drag source application is. Whichever way, the implementation of `ReleaseStgMedium` makes sure the space is freed by the correct party (see the Win32 API help for more details).

The remaining piece of the puzzle is to see how some different data formats are laid out in their respective storage media. Many of these data formats are documented, for example on the MSDN CD. Others, however, are undocumented and are for internal application use only. We'll look at a few of the project's data access methods (as space allows) to get the idea with some documented formats, starting with a simple one.

➤ *Table 1: Storage medium correlation.*

Storage Medium Type Constant	Storage Medium Record Data Field
TYMED_HGLOBAL	hGlobal
TYMED_FILE	lpszFileName
TYMED_ISTREAM	stm
TYMED_ISTORAGE	stg
TYMED_GDI	hBitmap
TYMED_MFPICT	hMetaFilePict
TYMED_ENHMF	hEnhMetaFile

```
PStgMedium = ^TStgMedium;
tagSTGMEDIUM = record
  tymed: Longint;
  case Integer of
    0: (hBitmap: HBitmap;
        unkForRelease:
        Pointer{IUnknown});
    1: (hMetaFilePict: THandle);
    2: (hEnhMetaFile: THandle);
    3: (hGlobal: HGlobal);
    4: (lpszFileName: POleStr);
    5: (stm: Pointer{IStream});
    6: (stg: Pointer{IStorage});
end;
TStgMedium = tagSTGMEDIUM;
STGMEDIUM = TStgMedium;
```

➤ *Listing 10: The TStgMedium record.*

CF_TEXT

When applications want to accept text dragged from another application, the `GetDataAsText` method is called. This takes a string `var` parameter which is meant to be assigned the dragged text. Listing 11 shows the call to the routine as well as its implementation.

Text data is stored in the clipboard as a standard C string (`PChar`) in a block of memory identified by a global memory handle. A pointer to the memory block can be obtained with a call to `GlobalLock` (`GlobalUnlock` must be called before finishing) and the C string can be translated to a Pascal string with a simple typecast.

CF_RTF

That was quite straightforward. What about more interesting data types? Rich text data is stored in the same way as normal text, so the `GetDataAsRTF` method looks very familiar. However, a rich edit control is quite capable of absorbing data from a data object on its own. All you need to do is pass the data object interface reference to the `ImportDataObject` method of the rich edit control's `IRichEditOle` interface. The `EM_GETOLEINTERFACE` message extracts the interface reference, as you can see in this month's *The Delphi Clinic*. Listing 12 shows what is necessary.

Also, if the drag operation causes the mouse to move over a rich edit control, you will find it already happy to accept the drop operation. To get the code added to the program doing the work, you must ensure you drag onto something other than a rich edit.

CF_HDROP

When files are dragged from Windows Explorer, as well as the `wm_DropFiles` message being sent to windows whose handles were passed to `DragAcceptFiles`, windows that were passed to `RegisterDragDrop` are sent a data object with the file list in the `CF_HDROP` format. What this means is that you can use the `hGlobal` field of the storage medium record as a `HDROP`, just as we did before. However, it is typical to rely on `ReleaseStgMedium` to free the memory, so the code in Listing 12 now becomes Listing 13.

➤ *Listing 11: Accessing dragged text data.*

```
//Code from the form
var Txt: String;
...
if dfText in DataObject.DataFormats then begin
  tsText.TabVisible := True;       //Show the text page
  DataObject.GetDataAsText(Txt);   //Get the text
  memText.Text := Txt;             //Give text to the memo
end;
...
//Code from TDataObject
procedure TDataObject.GetDataAsText(var Txt: String);
var
  SM: TStgMedium;
  CTxt: PChar;
begin
  SetupFormatEtc(CF_TEXT, TYMED_HGLOBAL);
  OleCheck(FDataObject.GetData(FFormatEtc, SM));
  try
    CTxt := GlobalLock(SM.hGlobal);
    try
      Txt := String(CTxt);
    finally
      GlobalUnlock(SM.hGlobal);
    end
  finally
    ReleaseStgMedium(SM)
  end
end;
```

➤ *Listing 12: Extracting dragged rich text.*

```
var
  RichEditOle: IRichEditOle;
...
if dfRTF in DataObject.DataFormats then begin
  tsRTF.TabVisible := True;
  reRTF.Lines.Clear;
  //Try and get richedit to deal with it...
  if reRTF.Perform(EM_GETOLEINTERFACE, 0, LParam(@RichEditOle)) <> 0 then
    RichEditOle.ImportDataObject(dataObj, 0, 0)
  else begin
    //If it can't do it yourself
    DataObject.GetDataAsRTF(Txt);
    reRTF.Lines.Text := Txt
  end;
end;
...
var CF_RTF: TClipFormat;
procedure TDataObject.GetDataAsRTF(var Txt: String);
var
  SM: TStgMedium;
  CTxt: PChar;
begin
  SetupFormatEtc(CF_RTF, TYMED_HGLOBAL);
  OleCheck(FDataObject.GetData(FFormatEtc, SM));
  try
    CTxt := GlobalLock(SM.hGlobal);
    try
      Txt := String(CTxt);
    finally
      GlobalUnlock(SM.hGlobal);
    end
  finally
    ReleaseStgMedium(SM)
  end
end;
initialization
  CF_RTF := RegisterClipboardFormat('Rich Text Format');
end.
```

CF_BITMAP and CF_PALETTE

Getting bitmaps and palettes from the clipboard (or from clipboard format) is quite straightforward thanks to some handy routines in the VCL's `Graphics` unit. As Listing 14 shows, the `TBitmap` class defines a method `LoadFromClipboardFormat` and there's a `CopyPalette` function.

CF_DIB

Whilst accessing a dragged bitmap is straightforward, getting a dragged DIB (device independent bitmap) requires extra tinkering. The DIB is accessible through a global memory handle. The DIB data is laid out like the majority of a bitmap file, but without a file header.

The code in Listing 15 sets up a suitable bitmap file header using a `TBitmapFileHeader` record (`BITMAPFILEHEADER` in the Windows API help). This record is then written to the beginning of a memory stream and the DIB data is added after it. The size of the DIB data can be easily learned with `GlobalSize`. With the memory stream now containing a whole bitmap file, it is suitable fodder for the `TBitmap` class's `LoadFromStream` method.

CF_ENHMETAFILE

Enhanced metafiles (EMFs) are easy to pick up (Listing 16): you just need to duplicate the metafile, an easy job with the `CopyEnhMetafile` API, the metafile handle can then be assigned to the `Handle` property of a metafile object.

CF_METAFILEPICT

However, normal Windows metafiles (WMFs) are trickier. The `TMetafile` class in 32-bit Delphi represents an enhanced metafile, not a normal metafile. This means that to get a dragged metafile into a `TMetafile` object, the Windows WMF must be converted into an EMF. Listing 17 shows the steps.

Firstly `GlobalLock` is used to turn the memory handle into a pointer to a `TMetaFilePict` record (`METAFILEPICT` in the Windows API help). This record contains the metafile handle along with size information and the original mapping mode (measurement system).

```
procedure TDataObject.GetDataAsHDrop(FileList: TStrings);
var
  SM: TStgMedium;
  Count, Loop: Integer;
  Buf: array[0..1023] of Char;
begin
  if not Assigned(FileList) then
    Exit;
  SetupFormatEtc(CF_HDROP, TYMED_HGLOBAL);
  OleCheck(FDataObject.GetData(FFormatEtc, SM));
  try
    //How many files were dragged?
    Count := DragQueryFile(SM.hGlobal,Cardinal(-1),nil,0);
    FileList.BeginUpdate;
    try
      FileList.Clear;
      //Loop through files
      for Loop := 0 to Pred(Count) do begin
        //Get filename
        DragQueryFile(SM.hGlobal, Loop, Buf, SizeOf(Buf));
        FileList.Add(Buf)
      end
    finally
      FileList.EndUpdate
    end
  finally
    ReleaseStgMedium(SM)
  end
end;
```

➤ *Listing 13: Accessing dragged files.*

A call to `GetMetaFileBitsEx` tells you how big the metafile is, so you can allocate a new buffer big enough to hold a copy of it. This duplicate metafile is then converted to an enhanced metafile with `SetWinMetaFileBits`, which returns the enhanced metafile handle.

Summary

Inter-application drag and drop can be added to an application without too much of a headache, so long as you take it one step at a time. Initialise the OLE library,

```
procedure TDataObject.GetDataAsBitmap(Bmp: TBitmap);
var SM: TStgMedium;
begin
  if not Assigned(Bmp) then Exit;
  SetupFormatEtc(CF_BITMAP, TYMED_GDI);
  OleCheck(FDataObject.GetData(FFormatEtc, SM));
  try
    //Use a handy shortcut to load bitmp
    Bmp.LoadFromClipboardFormat(CF_BITMAP, SM.hBitmap, 0);
    if dfPalette in DataFormats then GetDataAsPalette(Bmp)
  finally
    ReleaseStgMedium(SM)
  end
end;
procedure TDataObject.GetDataAsPalette(Bmp: TBitmap);
var SM: TStgMedium;
begin
  if not Assigned(Bmp) then Exit;
  SetupFormatEtc(CF_PALETTE, TYMED_GDI);
  OleCheck(FDataObject.GetData(FFormatEtc, SM));
  try
    Bmp.Palette := CopyPalette(SM.hBitmap)
  finally
    ReleaseStgMedium(SM)
  end
end;
```

➤ *Listing 14: Accessing a dragged bitmap and palette.*

implement support for `IDropTarget` and register the drop target window. When data is dropped, verify that the sort of data you are looking for exists and then get it. Once you are done with the dragged data, you free it. Before leaving the application, revoke drag/drop support and close down the OLE library.

The COMDragDrop.dpr project on the disk supports more formats than have been discussed in this article. Figure 2 shows the project running just after I dragged Figure 1 and its caption from a Microsoft Word document onto it. The metafile view is selected in the screenshot.

References

I have found a great reference for COM drag and drop (unfortunately for me, I had done most of my research for this article by the time I found it).

Grahame Marsh has written a series of articles on various drag/drop COM issues in *The Unofficial Newsletter of Delphi Users* (www.undu.com). The series, which started in Issue 31 of UNDU (August 1998), covers many aspects of this area that I am unable to fit in. These include getting the drop target to scroll as the cursor is moved towards the edges (like Windows Explorer), creating `Paste Special` dialogs, many more data formats and details on being a drag source.

Brian Long is a UK-based freelance consultant and trainer. He spends most of his time running Delphi and C++Builder training courses for his clients, and doing problem-solving work for them. Brian is at brian@blong.com


```
procedure TDataObject.GetDataAsDIB(Bmp: TBitmap);
var
  SM: TStgMedium;
  Stream: TMemoryStream;
  DIBPtr: Pointer;
  DIBSize: DWord;
  BMF: TBitmapFileHeader;
begin
  if not Assigned(Bmp) then
    Exit;
  SetupFormatEtc(CF_DIB, TYMED_HGLOBAL);
  OleCheck(FDataObject.GetData(FFormatEtc, SM));
  try
    DIBSize := GlobalSize(SM.hGlobal);
    DIBPtr := GlobalLock(SM.hGlobal);
    try
      Stream := TMemoryStream.Create;
      try
        //Write a bitmap file header record
        FillChar(BMF, sizeof(BMF), 0);
        BMF.bfType := $4D42;
        BMF.bfSize := SizeOf(BMF) + DIBSize;
        Stream.Write(BMF, SizeOf(BMF));
        Stream.Write(DIBPtr^, DIBSize);   //Follow the BMF with the DIB
        Stream.Position := 0;
        Bmp.LoadFromStream(Stream)        //Load the finished DIB into a TBitmap
      finally
        Stream.Free
      end
    finally
      GlobalUnlock(SM.hGlobal)
    end
  finally
    ReleaseStgMedium(SM)
  end
end;
```

➤ *Listing 15: Accessing a dragged DIB.*

```
procedure TDataObject.GetDataAsEMF(MetaFile: TMetafile);
var
  SM: TStgMedium;
begin
  if not Assigned(MetaFile) then
    Exit;
  SetupFormatEtc(CF_ENHMETAFILE, TYMED_ENHMF);
  OleCheck(FDataObject.GetData(FFormatEtc, SM));
  try
    MetaFile.Handle := CopyEnhMetafile(SM.hEnhMetaFile, nil)
  finally
    ReleaseStgMedium(SM)
  end
end;
```

➤ *Listing 16: Getting a dragged enhanced metafile.*

```
procedure TDataObject.GetDataAsWMF(MetaFile: TMetaFile);
var
  SM: TStgMedium;
  MPPtr: PMetaFilePict;
  MFBufSize: DWord;
  MFBuf: Pointer;
begin
  if not Assigned(MetaFile) then
    Exit;
  SetupFormatEtc(CF_METAFILEPICT, TYMED_MFPICT);
  OleCheck(FDataObject.GetData(FFormatEtc, SM));
  try
    MPPtr := GlobalLock(SM.hMetaFilePict);  //Get access to TMetaFilePict record
    try
      //How big is the metafile?
      MFBufSize := GetMetaFileBitsEx(MPPtr^.hMF, 0, nil);
      GetMem(MFBuf, MFBufSize);             //Allocate sufficient buffer space
      try
        //Copy metafile to buffer
        Win32Check(LongBool(GetMetaFileBitsEx(
          MPPtr^.hMF, MFBufSize, MFBuf)));
        //Generate enhanced metafile from buffer
        MetaFile.Handle := SetWinMetaFileBits(
          MFBufSize, MFBuf, 0, MPPtr^)
      finally
        //Free buffer
        FreeMem(MFBuf)
      end
    finally
      //Unlock memory handle
      GlobalUnlock(SM.hMetaFilePict)
    end
  finally
    ReleaseStgMedium(SM)
  end
end;
```

➤ *Listing 17: Accessing a dragged Windows metafile.*