Dragging And Dropping Part 1: VCL

by Brian Long

The Delphi VCL has supported drag and drop operations ever since version 1, way back in 1995. It was enhanced just a little in the first 32-bit version (Delphi 2 in 1996) but, apart from that, the basic mechanisms have remained much the same.

Whilst using drag and drop in a Delphi application is made very easy by the VCL support, little seems to have been written on the subject, which makes doing more interesting variations on the standard theme more complicated.

This article is the first in a series of three that will be looking at the subject of drag and drop in Delphi applications. This one will look at the basic VCL support for dragging and dropping within a Delphi application.

Next month's instalment looks at how custom drag objects can be used to enhance the appearance of drag operations, and how they can simplify complex drag operations.

The final part of this mini-series will look at the more involved subject of dragging and dropping between applications (requiring us to do COM things and make use of many esoteric constants and record structures).

Here Thar Be Drag (ons)

The VCL has in-built facilities for supporting dragging and dropping within a given application. A drag and drop operation relies upon the user clicking the left (or sometimes the right) mouse button down on some control, then moving their mouse (whilst keeping the button held down) over to another control, and finally releasing the mouse button.

The control that initially gets clicked on is called the *source* of the drag, or *drag source*, and the one under the mouse when the button is released is called the *target*, the *drag target* or *drop target*. As far as the user is concerned, they are under the impression that they are physically dragging the source control (or information shown on it) onto the target control. In truth of course, the user is just moving the mouse with a button held down, however the terminology used (and maybe the cursor image displayed) upholds the user's view.

It is down to the application to interpret this particular mouse operation and do something sensible with it. This includes changing the mouse cursor to suggest a drag operation is occurring, as well as indicating whether the control under the mouse is happy to accept the dragged source control or not. Of course it also involves doing something when the user ends a drag operation by dropping the dragged control on another control.

Fortunately the VCL makes short shrift of these requirements. The tricky stuff is dealt with by the code in the `Controls` unit sending internal component messages around under appropriate circumstances (the `cm_Drag` message, with various parameters). As far as the Delphi programmer is concerned, there are three simple steps to get drag and drop working.

Firstly, you must enable dragging from the source control. Then, when someone starts dragging from the source, a drag operation will be started. The VCL will set the mouse cursor appropriately as you move your mouse around the screen. By default, it will be a 'No Entry' type cursor. You can see this cursor by setting a control's `Cursor` property to `crNoDrop`, which looks like .

The next step is to make the potential target controls indicate that they are happy to accept the source being dropped. At runtime, this is indicated by the cursor changing to a normal drag cursor (the `TCursor` type value of `crDrag` by default, which looks like). This default cursor is obtained from the dragged control's `DragCursor` property.

The final step is to implement what happens if the user drops the source control onto the target control. By default, nothing happens except the drag operation is terminated.

Let's go through these steps one at a time, looking at what possibilities the VCL offers.

What A Drag!

Normally, when you click and drag on an arbitrary control on a form, nothing particularly special happens. Specifically, no drag operation starts off (no cursor changes, no ability to drop, etc).

You can cause a drag operation to start in one of two ways: automatically or manually. To start drag operations automatically, set the source control's `DragMode` property to `dmAutomatic` (it defaults to `dmManual`). This property is defined in `TControl`, so all visual controls will have it, be they Delphi versions of real Windows controls or not. The effect of setting `DragMode` to `dmAutomatic` is that clicking the left mouse button down on a control will automatically start a drag operation without any extra code.

Without using this `DragMode` setting, you can start a drag operation manually with a call to the control's `BeginDrag` method. `BeginDrag` has one mandatory parameter, which is a `Boolean` that dictates whether the drag operation begins immediately. One reason for calling `BeginDrag` is that you might want to only allow drag operations under some special circumstances. Take an edit or memo control for example. These already use mouse dragging operations to highlight text. If you want to permit text selection as well as allow dragging from the edit/memo, you could restrict drag operations to only start when the Control key is held down, for example. Listing 1 shows how an edit control's OnMouseDown event handler could achieve this.

Alternatively, you may want to start a drag operation with the right mouse button, rather than the left (Windows dragging supports the right mouse button, but normal VCL dragging is only invoked by the left mouse button). Listing 2 shows how an event handler might do this.

A value of True passed to BeginDrag immediately starts a drag operation. On the other hand, a value of False will only start the full drag operation when the mouse moves a certain number of pixels from where it was clicked. The point of this is for normal clicks to be permitted (where the mouse doesn't move, thereby not causing the drag to start), whilst still allowing dragging operations (which only kick in when the mouse moves a certain number of pixels).

For example, a listbox control could use the code shown in Listing 3 as an OnMouseDown event handler. This would allow you to left click on items in the list as usual, without starting drag operations, and would also allow you to start a drag operation by left clicking on an item (not on a blank area) and dragging the mouse.

When False is passed to BeginDrag, the user must move the mouse 5 pixels to start a drag. In Delphi 1, 2 and 3 this is a fixed value, but Delphi 4 (and later) allows you to specify alternative pixel distances. You can change the default non-immediate mouse drag distance threshold by assigning a new value to Mouse.DragThreshold (Mouse is a global object instance, created in the Controls unit). Alternatively, you can pass an optional second parameter to BeginDrag. This parameter defaults to -1, meaning that Mouse.DragThreshold will be used.

A value of dmAutomatic assigned to the DragMode property of a Delphi 1 to Delphi 3 control causes the control to call BeginDrag(True) when the left mouse button is clicked on it (that is, an immediate drag operation starts). Delphi 4 and later changes this. The control now calls the protected polymorphic BeginAutoDrag method (declared as dynamic), which calls:

```
BeginDrag(Mouse.DragImmediate,
  Mouse.DragThreshold).
```

You can change the values of Mouse.DragImmediate and Mouse.DragThreshold to affect global drag operations (despite the online help currently suggesting these properties are read-only). Custom components can also override the BeginAutoDrag method to change what happens if the user drags them when DragMode is set to dmAutomatic. In fact TCustomForm overrides it and does nothing in the re-implementation, to ensure that the user cannot drag a form if its DragMode property is set to dmAutomatic.

Will You Accept This Object?

After allowing some drag operations to start, using one of the two ways described above, we now need to get some other target control (or controls) to indicate that they will accept something dragged from the source. This is done by writing an OnDragOver event handler for the target control(s) (an empty one is shown in Listing 4).

The event handler has a number of parameters that give information on the drag operation.

The most important parameter is Accept, which is a var parameter. You set this to False if you do not accept a drag from the suggested source. It defaults to True, which means that by default, an OnDragOver event handler will accept any source. However, if you do not make an OnDragOver event

```
procedure TForm1.Edit1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  //Check this is an edit and Ctrl is pressed
  if (Sender is TCustomEdit) and (ssCtrl in Shift) then
    TCustomEdit(Sender).BeginDrag(True)
end;
```

➤ *Listing 1: Allowing dragging from an edit control, without affecting text selection.*

```
procedure TForm1.Edit1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbRight then
    (Sender as TControl).BeginDrag(False)
end;
```

➤ *Listing 2: Starting a drag operation with the right mouse button.*

➤ *Listing 3: Manually starting a listbox drag operation.*

```
procedure TForm1.ListBox1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  //Check this is a listbox left mouse button event
  if (Sender is TCustomListBox) and (Button = mbLeft) then
    with TCustomListBox(Sender) do
      //Verify mouse is over a listbox item
      if ItemAtPos(Point(X, Y), True) <> -1 then
        //Start a non-immediate drag operation
        BeginDrag(False)
end;
```

➤ *Listing 4: An OnDragOver event handler.*

```
procedure TForm1.ImageDragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
end;
```

```
procedure TForm1.Memo1DragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if Source is TEdit then
    Accept := True
  else
    Accept := False
  //The above can be written more succinctly as:
  // Accept := Source is TEdit
end;
```

➤ *Listing 5: A simple OnDragOver event handler.*

handler, the control will not accept any drag operations.

You can see a simple OnDragOver event handler in Listing 5. This is an OnDragOver event handler for a memo component, that will accept something dragged only from an edit control.

The effect of the Accept parameter being set to True is that the usual No Entry drag cursor will change to the dragged control's DragCursor property when the mouse is over a target control that accepts it. The user will be allowed to drop the dragged control on the target, although at this point nothing will happen when they do so.

Sender represents the control whose event is firing, and this is happening because the control indicated by the Source parameter is currently over it at the position indicated by the X and Y parameters, relative to Sender.

The State parameter tells how the mouse is moving relative to the control under the mouse. As a dragging operation proceeds, when the mouse enters a control, its OnDragOver event is triggered with State set to dsDragEnter. It is also repeatedly triggered as the mouse moves over the control (State is dsDragMove) and potentially triggered one last time when the mouse moves out of a control, or the drag operation is terminated whilst the mouse is over the control (State is dsDragLeave).

You can use the State parameter to start certain operations, allocate various resources or whatever, as the user starts dragging across a given control. You can then stop the operation, or free the resources when the user drags the mouse out of the control, or the drag operation is terminated whilst over that control.

Listing 6 shows a simple (if entirely academic) application of this. Assuming the target component (a memo) is happy to accept the source (an edit), information about the drag operation is displayed in a label whilst the mouse is moved over the target. At any point when the drag operation is not active, or when the target control is not the memo, the label is invisible.

OnDragOver is called, if present, from the DragOver dynamic protected method. Custom component classes can override this method to provide additional functionality when a drag operation moves over them, if necessary.

Dropping Off...

When the user eventually performs the 'drop' part of the drag and drop operation, by releasing the mouse over a target control that claims to accept it, the target control's OnDragDrop event handler is invoked. Listing 7 shows an empty OnDragDrop event handler.

The parameters are a subset of the OnDragOver event handler's parameters. The user dragged the Source control and dropped it on Sender. X and Y are the co-ordinates relative to the control that was dropped on (Sender).

The code in an OnDragDrop event handler can do whatever is necessary to implement the drop. If the control in question can accept drops from multiple sources, acting differently for each one, then some more checking of the Source parameter will be required.

The OnDragDrop event handler is called from the public DragDrop dynamic method. Component classes can potentially perform custom drop functionality in an overridden version of this method. Being public, you could also get the same behaviour as a drag and drop operation by directly calling a target control's DragDrop method, passing the source object and X and Y co-ordinates. For example, this statement gets the same end result as the user initiating a drag operation from an edit control and dropping it on a memo:

```
Memo1.DragDrop(Edit1, 0, 0);
```

Testing The Theory

Having got through the three basic steps for drag and drop, let's build a simple application that employs drag 'n' drop. It will be built first of all with no drag and drop support, and then we will retro-fit drag and drop support into the application.

The application will allow the user to navigate around their drives, using some of the older navigation components that were

➤ *Listing 6: An OnDragOver event handler that uses the State parameter.*

```
procedure TForm1.Memo1DragOver(Sender, Source: TObject; X, Y: Integer;
  State: TDragState; var Accept: Boolean);
begin
  Accept := Source is TEdit;
  if Accept then
    case State of
      dsDragEnter : Label1.Show;
      dsDragMove  : Label1.Caption := Format('Dragging %s to %s at (%d,%d)',
        [(Source as TControl).Name, (Sender as TControl).Name, X, Y]);
      dsDragLeave: Label1.Hide
    end;
end;
```

➤ *Listing 7: An OnDragDrop event handler.*

```
procedure TForm1.ImageDragDrop(Sender, Source: TObject; X, Y: Integer);
begin
end;
```

on the System page of the Delphi 1 Component Palette but are on the Win 3.1 page of all later versions. The user can employ these components to locate bitmap files on their machine. When one or more bitmap files are shown in the file listbox, double clicking one of them will load it into an image component, also on the form. The image component is surrounded by a bevel component, to make the image's location more obvious.

Add the following key components to the form of a new application, changing their names to what is enclosed in brackets: a TLabel (DirLabel), a TImage (Image), a TEdit (FileEdit), a TFileListBox (FileList), a TDirectoryListBox (DirList), a TDriveComboBox (DriveCombo), and a TFilterComboBox (FilterCombo). Apart from positioning and sizing the components, the important properties should be set as shown in Listing 8.

Now make an OnDblClick event handler for the file listbox as per the code in Listing 9, which loads the selected bitmap file into the image control (assuming it is in a supported format).

This gives us an application that has no support for drag and drop, but which does allow bitmap files to be loaded into an image component. So now we can add the important drag and drop support with the three previously outlined steps.

The first step is to enable the drag from the source (the file listbox). This can be done simply by setting the DragMode property to dmAutomatic, which means any left click anywhere on the listbox will start a drag operation. Alternatively, you can make an OnMouseDown event handler with code like that shown in Listing 3. Since a TFileListBox is indirectly inherited from TCustomListBox, the same code will work fine.

```
object Image: TImage
  Stretch = True
end
object FileEdit: TEdit
  Text = '*.bmp'
end
object FileList: TFileListBox
  FileEdit = FileEdit
end
object DirList: TDirectoryListBox
  DirLabel = DirLabel
  FileList = FileList
end
object DriveCombo: TDriveComboBox
  DirList = DirList
end
object FilterCombo: TFilterComboBox
  FileList = FileList
  Filter =
    'Bitmap files (*.bmp)|*.bmp|All files (*.*)|*.*'
end
```

➤ *Listing 8: Property values for the first drag and drop application.*

```
procedure TForm1.FileListDblClick(Sender: TObject);
begin
  Image.Picture.LoadFromFile(FileList.FileName)
end;
```

➤ *Listing 9: Loading a file into an image component.*

The second step is to tell the image component to accept anything dragged from the file listbox. This involves making an OnDragOver event handler for the image component with the following logical assignment within it:

```
Accept := Source = FileList
```

Finally, when the user drops on the image component we need to load the file as selected in the file listbox into the image. This requires an OnDragDrop event handler for the image component. The statement in the file listbox's OnDblClick event handler (Listing 9) could be duplicated in the new event handler, but code duplication is usually a bad thing, and is to be avoided. Besides, in a real application, the code that would need duplicating might be considerably larger.

Instead, we will invoke the file listbox's OnDblClick event handler from within the image's OnDragDrop event handler. You can do this directly, as in:

```
FileListDblClick(FileList);
```

or, in 32-bit Delphi, you can do it indirectly, by referring to the event property of the component in question, as shown in Listing 10. In both cases, I am ensuring that the file listbox is passed as the Sender parameter to the event handler, just in case the event handler makes use of that parameter. In an event handler, Sender should always refer to the object whose event is being handled. Delphi 1 does not support the syntax used to verify that an event property has an event handler associated with it.

The application is now complete, and you can find a copy on this month's disk as VCLDrop.Dpr. Run it in any version of Delphi, and you should find that you can drag a bitmap file from the file listbox onto the image component, which will then load the bitmap and display it. You can see a file being dragged onto the image component in Figure 1.

➤ *Listing 10: Invoking the file listbox's OnDblClick event handler.*

```
procedure TForm1.ImageDragDrop(Sender, Source: TObject; X, Y: Integer);
begin
  { If FileList has an OnDblClick event handler... }
  if Assigned(FileList.OnDblClick) then
    { ... invoke it }
    FileList.OnDblClick(FileList)
end;
```

Customising Drag Operations

The VCL has a number of routines up its metaphorical sleeve that can be used to analyse and customise drag and drop operations.

As has been mentioned, when a dragged control is over a target control that will accept it, the mouse cursor is set to the dragged control's `DragCursor` property. This property defaults to `crDrag`, but you can change it to other values to modify the drag cursor appearance. You can either choose one of the pre-defined system cursors, or use a custom mouse cursor. The Cursor.Dpr project uses a custom drag cursor, as shown in Figure 2.

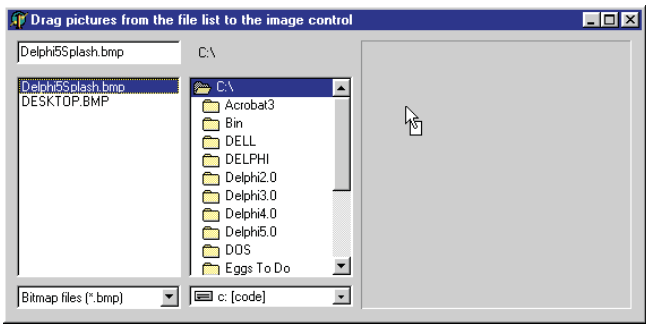
To load a custom mouse cursor, make a Windows resource file containing the cursor (using Resource Workshop, or the Image Editor that comes with Delphi, or some other tool if you prefer). A sample cursor resource file is on this month's disk (PacCur16.Res for Delphi 1 and PacCur32.Res for all 32-bit versions) containing a cursor named *PacMan*. The code in the program required to load this custom cursor into a control's `DragCursor` property is shown in Listing 11.

All controls have a public `Dragging` method. This parameterless function returns `True` if the control is being dragged (which means that a drag operation was initiated through that control, and has not yet terminated). This allows any piece of code (not just the code in `OnDragOver` and `OnDragDrop` event handlers) to check whether a certain control is currently in the process of being dragged.

To complement the `BeginDrag` method, controls also have an `EndDrag` method that allows you to programmatically terminate a drag operation. `EndDrag` takes a `Boolean` parameter called `Drop`. If `Drop` is `True` and the mouse is over a control that will accept the drag, then

➤ *Listing 11: Loading a custom drag cursor from a Windows resource file.*

```
const
  crPacMan = 1; { Use values > 0 }
...
{$ifdef Windows}
  {$R PacCur16.Res}
{$else}
  {$R PacCur32.Res}
{$endif}
procedure TForm1.FormCreate(
  Sender: TObject);
begin
  Screen.Cursors[crPacMan] :=
    LoadCursor(
      HInstance, 'PacMan');
  Edit1.DragCursor := crPacMan
end;
```



➤ *Figure 1: An application that supports drag and drop.*

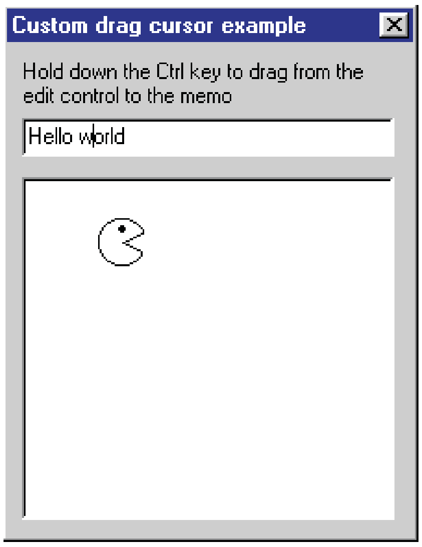
the control being dragged is dropped. Under all other circumstances, the drag operation is cancelled.

More generically, the `Controls` unit (in Delphi 2 and later) implements a `CancelDrag` procedure which cancels the current drag operation, if there is one, without dropping the dragged object.

A drag operation can therefore be terminated in a number of ways. The user can positively terminate a drag by dropping the control on a target that accepts it. They can also cancel the drag by dropping the control on something that does not accept it, or by pressing the `Escape` key. The programmer can terminate the drag positively or negatively using the dragged control's `EndDrag` method, or cancel it with `CancelDrag`.

If a custom component needs to do anything particularly special when a drag operation is cancelled, it can override the protected dynamic method `DragCanceled`. By default, this does nothing. However, the `TCustomListBox` class overrides it to fix certain mouse usability issues that arise when a drag operation is cancelled.

The `Controls` unit offers another global routine called `FindDragTarget`. This takes a `TPoint` record that describes a screen location and is designed to return the control that occupies that position. Whilst its name suggests it will return a target control that is ripe for accepting things, it does no such checking. It will return the control at the specified screen position, and that control may or may not have suitable `OnDragOver` and `OnDragDrop` event handlers. The only extra checking performed by this routine is dictated by the additional `Boolean` parameter (`AllowDisabled`) that controls whether disabled controls will be considered for returning. If no


Custom drag cursor example
Hold down the Ctrl key to drag from the edit control to the memo
Hello world

➤ *Figure 2: Dragging from an edit to a memo with a custom drag cursor.*

control can be found at the specified position, `FindDragTarget` returns `nil`.

Whilst the `OnDragOver` event handler's `State` parameter can enable you to start operations when the user drags one control into another one, and then stop those operations when the control is dragged back out, there are two events that allow you to do more widespread operations.

A control's `OnStartDrag` event handler (see Listing 12) will be triggered as soon as a drag operation on it starts, either through a call to its `BeginDrag` method, or by being clicked on when `DragMode` is set to `dmAutomatic`. We will look more closely at what we can do with this event, which was introduced in Delphi 2, in next month's instalment.

A corresponding `OnEndDrag` event handler is called when the drag operation stops (also shown in Listing 12). This can either be because the control was dropped,

```
procedure TForm1.Label1StartDrag(Sender: TObject; var DragObject: TDragObject);
begin
end;
procedure TForm1.Label1EndDrag(Sender, Target: TObject; X, Y: Integer);
begin
end;
```

➤ *Listing 12: OnStartDrag and OnEndDrag event handlers.*

or because the operation was terminated in some way. This event (which has been around since Delphi 1) takes four parameters. The ever-present `Sender` parameter is the control that is no longer being dragged. `Target` represents the control that `Sender` was dropped on, but which can be `nil` in the case of a terminated drag operation. The `X` and `Y` co-ordinates, relative to `Target`, are also passed as parameters, though if `Target` is `nil` these parameters will both be `0`.

The `OnEndDrag` event is triggered after execution of the `DragCanceled` method. If the drag is terminated successfully with a drop, the source's `OnEndDrag` event occurs after the target's `OnDragDrop` event.

`OnStartDrag` and `OnEndDrag` are called from the protected dynamic methods `DoStartDrag` and `DoEndDrag` respectively, which again can be overridden by new component classes to perform additional tasks specific to the component being written.

Summary

This article has endeavoured to describe the rich VCL support for easy drag and drop in your applications. Whilst it is very flexible and customisable, all you need to start with is three simple steps to add drag and drop support into your application.

Next month we will look at how custom drag objects can be used to both enhance the appearance of the mouse cursor when a control is being dragged, and also how they can simplify the coding of more complex drag operations.

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. You can reach him at brian@blong.com