COM Corner: Automation Events

by Steve Teixeira

We Delphi programmers have long taken events for granted. You drop a button, you double click on `OnClick` in the Object Inspector, and you write some code. No big deal. Even from the control writer's point of view, events are a snap. You create a new method type, add a field and `published` property to your control, and off you go. For Delphi COM developers, however, events can be scary. Many Delphi COM developers avoid events altogether. If you fall into that group, or if you're one of the brave ones that implemented events in your Automation servers despite the lack of built-in support, you'll be happy to know that working with events has become much easier with Delphi 4. While all of the new terms associated with Automation events can add an air of complexity, in this article I hope to de-mystify events to the point where you think, 'Oh, is that all they are?'

What Are Events?

Put simply, events provide a means for a server to call back into a client to provide some information. Under a traditional client/server model, the client calls the server to perform an action or obtain some data, the server executes the action or obtains the data, and passes control to the client. This model works fine for most things, but it breaks down when the event in which the client is interested is asynchronous in nature or is driven by user interface entry. For example, if the client sends the server a request to download a file, the client probably doesn't want to sit around and wait for the thing to download before it can continue processing (especially over a high latency connection like a modem). A better model would be for the client to issue the instruction to the server and continue to go about its business until the server notified the client about the completion of the file download. Similarly, user interface entry, like clicking a button, is a good example of a case when the server needs to notify the client using an event mechanism. The client obviously can't call a method on the server that waits around until a button is clicked.

Generally speaking, the server is responsible for defining and firing events, while the client is normally responsible for connecting itself to and implementing events. Of course, given such a loose definition, there is room to haggle and consequently Delphi and Automation provide two very different approaches to events. A little drill-down into each of these models will put things into perspective.

Events In Delphi

Delphi follows the KISS (keep it simple, stupid!) methodology when it comes to events. Events are implemented as method pointers: these pointers can be assigned to some method in the application, and are executed when such a method is called via the method pointer. As an illustration, consider the everyday scenario of an application that needs to handle an event on a component. If you look at the situation abstractly, the 'server' in this case would be a component, which defines and fires the event. The 'client' is the application employing the component, since it connects to the event by assigning some specific method name to the event method pointer.

While this simple event model is one of the things that make Delphi elegant and easy to use, it definitely sacrifices some power for the sake of usability. For example, there is no built-in way to allow multiple clients to listen for the same event (this is called multicasting). Also, there is no way to dynamically obtain a type description for an event without writing some really RTTI code (that you probably shouldn't be using in an application anyway, due to its version-specific nature).

Events In Automation

While the Delphi event model is simple yet limited, the Automation event model is powerful but more complex. As you may have guessed, events are implemented in Automation using interfaces. Rather than existing on a per-method basis, events exist only as part of an interface. This interface is often called an *events interface* or an *outgoing interface*. It's called 'outgoing' because it is not implemented by the server, like other interfaces, but is instead implemented by clients of the server, and methods of the interface will be called outward from server to client. Like all interfaces, event interfaces have associated with them a corresponded Interface Identification (IID) that uniquely identifies them. Also, the description of the events interface is found in the type library of an Automation object, tied to the Automation object's coclass like other interfaces.

Servers wishing to surface event interfaces to clients must implement the `IConnectionPointContainer` interface. This interface is defined in the `ActiveX` unit as shown in Listing 1.

In COM parlance, a *connection point* is a term that describes the entity that provides programmatic access to an outgoing interface. If a client wants to check if a server supports events, all it has to do is `QueryInterface` for the `IConnectionPointContainer` interface. If this interface is present the server is capable of surfacing events. The `EnumConnectionPoints` method of `IConnectionPointContainer` enables clients to iterate through the outgoing interfaces supported by the server. Clients use the `FindConnectionPoint` method to obtain a specific outgoing interface.

```
type
  IConnectionPointContainer = interface
    ['{B196B284-BAB4-101A-B69C-00AA00341D07}']
    function EnumConnectionPoints(out Enum: IEnumConnectionPoints): HResult;
      stdcall;
    function FindConnectionPoint(const iid: TIID; out cp: IConnectionPoint):
      HResult; stdcall;
  end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
type
  IConnectionPoint = interface
    ['{B196B286-BAB4-101A-B69C-00AA00341D07}']
    function GetConnectionInterface(out iid: TIID): HResult; stdcall;
    function GetConnectionPointContainer(out cpc: IConnectionPointContainer):
      HResult; stdcall;
    function Advise(const unkSink: IUnknown; out dwCookie: Longint): HResult;
      stdcall;
    function Unadvise(dwCookie: Longint): HResult; stdcall;
    function EnumConnections(out Enum: IEnumConnections): HResult; stdcall;
  end;
```

You'll notice that `FindConnectionPoint` provides an `IConnectionPoint` which represents an outbound interface. `IConnectionPoint` is also defined in the `ActiveX` unit, and it looks like Listing 2.

The `GetConnectionInterface` method of `IConnectionPoint` provides the IID of the outgoing interface supported by this connection point. The `GetConnectionPointContainer` method provides the `IConnectionPointContainer` (described above) which manages this connection point. The `Advise` method is the interesting one: it actually does the magic of hooking up the outgoing events on the server to the events interface implemented by the client. The first parameter to this method is the client's implementation of the events interface and the second parameter will receive a cookie that identifies this connection. `Unadvise` simply disconnects the client/server relationship established by `Advise`. `EnumConnections` enables the client to iterate over all active connections (that is, all connections that have called `Advise`).

Because of the obvious confusion that can arise if we describe the participants in this relationship as simply client and server, Automation defines some different terminology. The implementation of the outgoing interface contained within the client is called a *sink*, and the server object which fires events to the client is referred to as the *source*.

What is hopefully clear in all this is that Automation events have a couple of advantages over Delphi events. Namely, they can be multicast, since `IConnectionPoint.Advise` can be called more than once. Also, Automation events are self-describing (via the type library and the enumeration methods), so they can be manipulated dynamically.

Delphi 4 Automation Events

Okay, all that technical stuff is well and good, but how do we actually make Automation events work in Delphi? Glad you asked. At this point, I'll create an Automation server application that exposes an outgoing interface and a client which implements a sink for the interface. Bear in mind, too, that you don't need to be an expert in connection points, sinks, sources, and whatnot in order to get Delphi to do what you want. But it does help you in the long run when you understand what goes on behind the Wizard's curtain.

The Server

The first step in creating the server is to create a new application. For this demo I will create a new application with one form, with a client-aligned `TMemo`, as shown in Figure 1.

Next, I add an Automation object to this application by selecting `File | New | ActiveX | Automation Object` from the main menu. This invokes the Automation Object Wizard as shown in Figure 2.

You might notice the `Generate Event Support Code` option is new to Delphi 4. This box must be selected, as it will generate the code necessary to expose an outgoing interface on the Automation object. It will also create the outgoing interface in the type library. After selecting `Ok` in this dialog, I am presented with the Type Library Editor window. Both the Automation interface and the outgoing interface are already present in the type library (named `IServerWithEvents` and `IServerWithEventsEvents`, respectively). I added several new methods to the interfaces, which can be seen in Figure 3.

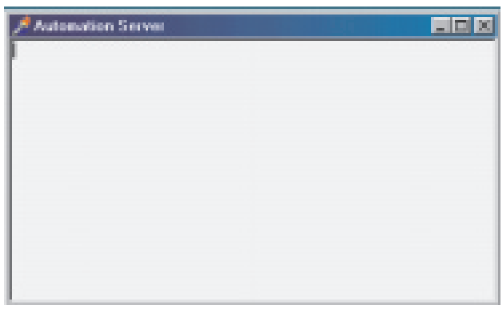
As you might guess, `Clear` will clear the contents of the memo and `AddText` will add another line of text to the memo. The `OnTextChanged` event will fire when the contents of the memo change, and the `OnClear` event will fire when the memo is cleared. Notice also that `AddText` and `OnTextChanged` each have one parameter of type `WideString`.

The first thing to do is implement the `AddText` and `Clear` methods. The implementation for these methods is shown in Listing 3.

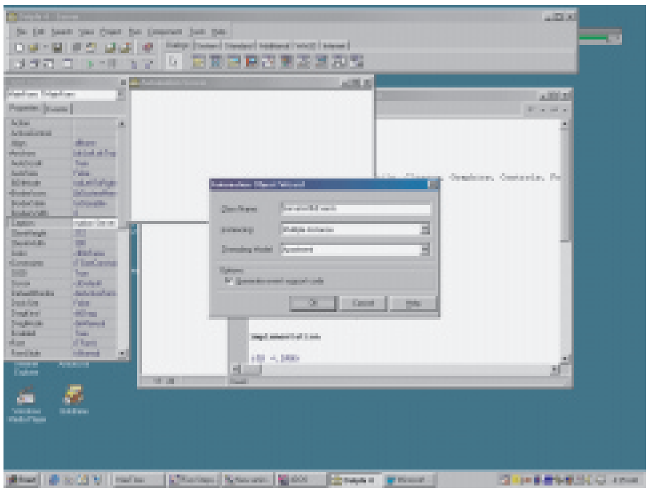
I expect you are familiar with all of the above code except the last line of `Clear`. This code checks to ensure that there is a client sink on the event by checking for `nil`, then first fires the even simply by calling `OnClear`.

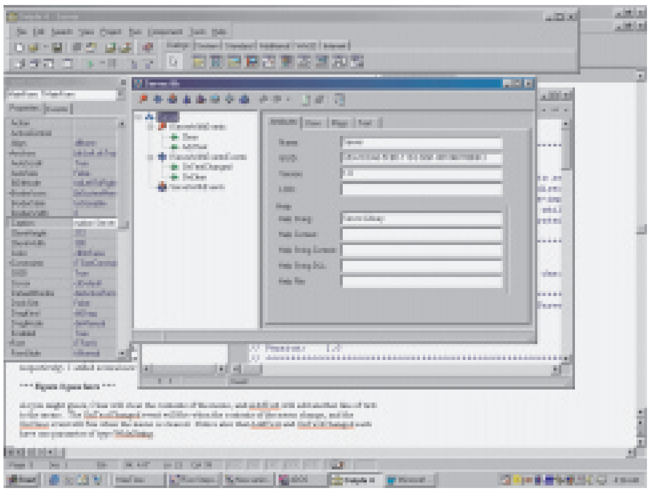
To set up the `OnTextChanged` event, I first have to handle the `OnChange` event of the memo. I do this by inserting a line of code into the Initialized method of `TServerWithEvents` which points the event to my own method in `TServerWithEvents`:

```
MainForm.Memo.OnChange :=
  MemoChange;
```



➤ *Figure 1*





➤ *Left: Figure 2*
➤ *Right: Figure 3*

```
procedure TServerWithEvents.AddText(const NewText: WideString);
begin
  MainForm.Memo.Lines.Add(NewText);
end;
procedure TServerWithEvents.Clear;
begin
  MainForm.Memo.Lines.Clear;
  if FEvents <> nil then FEvents.OnClear;
end;
```

➤ *Listing 3*

My `MemoChange` method is implemented as shown in Listing 4.

This code also checks to ensure a client is listening, then fires the event, passing the memo's text as the parameter.

Believe it or not, that sums it up for the implementation of the server! Now on to the client...

The Client

The client is an application with one form, which contains a `TEdit`, `TMemo`, and three `TButtons`, as shown in Figure 4.

In the main unit for the client application, I add the `Server_TLB` unit to the `uses` clause so that I have access to the types and methods contained within that unit. The main form object, `TMainForm`, of my client application will contain a field which references the server called `FServer` of type `IServerWithEvents`. I will create an instance of the server in `TMainForm`'s constructor using the helper class found in `Server_TLB` like this:

```
FServer :=
  CoServerWithEvents.Create;
```

Next step is to implement the event sink class. Since this class will be called by the server via Automation, it must implement `IDispatch` (and therefore `IUnknown`). The type declaration for this class is shown in Listing 5.

Most of the methods of `IUnkown` and `IDispatch` aren't implemented, with the notable exception of `IUnknown.QueryInterface` and `IDispatch.Invoke`. We will discuss these in turn.

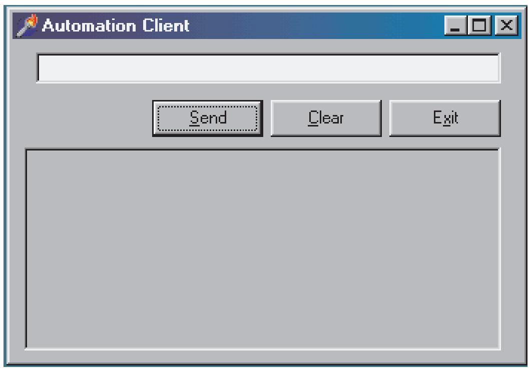
The `QueryInterface` method for `TEventSink` is implemented as in Listing 6.

Essentially, this method returns an instance only when the requested interface is `IUnknown`, `IDispatch` or `IServerWithEventsEvents`. The Invoke method for `TEventSink` is shown in Listing 7.

`TEventSink.Invoke` is hard-coded for methods having DispID 1 or DispID 2, which happen to be the DispIDs I chose for `OnTextChanged` and `OnClear` respectively in my server application. `OnClear` has the most straightforward implementation: it simply calls the client main form's `OnClear` method in response to the event. The `OnTextChanged` event is a little trickier: this code pulls the parameter out of the `Params.rgvarg` array, which is passed in as a parameter to this method, and passes it through to the client main form's `OnServerMemoChanged` method. Note that since I knew the number and type of parameters, I was able to make simplifying assumptions in my source code. If you're clever, it is possible to implement `Invoke` in a generic manner such that it figures out the number and types of parameters and pushes them onto the stack and/or into registers prior to calling the appropriate function. If you'd like to see an example of this, take a look at the `TOleControl.InvokeEvent` method in the `OleCtrls` unit. That method represents the event sinking logic for the ActiveX control container.

The implementation for `OnClear` and `OnServerMemoChanged` manipulate the contents of the client's memo, and they are shown in Listing 8.

The final piece of the puzzle is to connect the event sink to the server's source interface. This is easily accomplished using the `InterfaceConnect` function found in



➤ *Figure 4*

the `ComObj` unit, which I call from the main form's contructor like so:

```
InterfaceConnect(FServer,
  IServerWithEventsEvents,
  FEventSink, FCookie);
```

The first parameter to this function is a reference to the source object. Parameter two is the IID of the outgoing interface. The third parameter holds the event sink interface. The fourth and final parameter is the cookie, and it is a reference parameter that will be filled in by the callee.

To be a good citizen, you should also clean up properly by calling `InterfaceDisconnect` when you are finished playing with events. I do this in the main form's destructor:

```
InterfaceDisconnect(FEventSink,
  IServerWithEventsEvents,
  FCookie);
```

The Demo

Now that the client and server are written, we can see them in action. Be sure to run and close the server once (or run with the `/regserver` switch) to ensure it is registered before attempting to run the client. Figure 5 shows the interactions between client, server, source and sink in living color.

Summary

We covered a lot of ground in a short amount of time, including events in general, Delphi and Automation events specifically, and how to use these events in your Automation applications. Hopefully, Automation events make a little more sense now. More importantly, I hope you now have a better idea how to use Automation events in your own Delphi applications.

Steve Teixeira is Director of Software Development at DeVries Data Systems, a consulting firm specializing in Delphi development, and co-author of *Delphi 4 Developer's Guide*. If you have a COM question that you would like to see answered in this column, email Steve at steve@dvdata.com

```
procedure TServerWithEvents.MemoChange(Sender: TObject);
begin
  if FEvents <> nil then FEvents.OnTextChanged((Sender as TMemo).Text);
end;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
type
  TEventSink = class(TObject, IUnknown, IDispatch)
  private
    FController: TMainForm;
    { IUnknown }
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
    { IDispatch }
    function GetTypeInfoCount(out Count: Integer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo):
      HResult; stdcall;
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HResult; stdcall;
    function Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer; Flags:
      Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer): HResult; stdcall;
  public
    constructor Create(Controller: TMainForm);
  end;
```

```
function TEventSink.QueryInterface(const IID: TGUID; out Obj): HResult;
begin
  // First look for my own implementation of an interface
  // (I implement IUnknown and IDispatch).
  if GetInterface(IID, Obj) then
    Result := S_OK
  // Next, if they are looking for outgoing interface, recurse to return
  // our IDispatch pointer.
  else if IsEqualIID(IID, IServerWithEventsEvents) then
    Result := QueryInterface(IDispatch, Obj)
  else
    Result := E_NOINTERFACE;  // For everything else, return an error.
end;
```

➤ *Above: Listing 6*

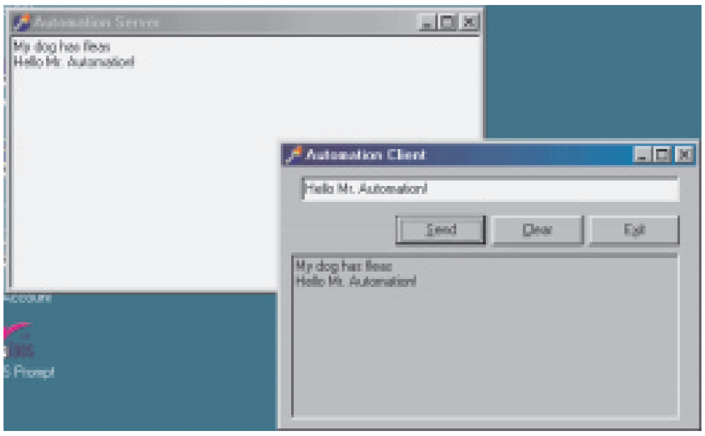
➤ *Below: Listing 7*

```
function TEventSink.Invoke(DispID: Integer; const IID: TGUID;LocaleID: Integer;
  Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer): HResult;
var V: OleVariant;
begin
  Result := S_OK;
  case DispID of
    1 : begin
          V := OleVariant(TDispParams(Params).rgvarg^[0]); // new string
          FController.OnServerMemoChanged(V);
        end;
    2 : FController.OnClear;
  end;
end;
```

```
procedure TMainForm.OnServerMemoChanged(const NewText: string);
begin
  Memo.Text := NewText;
end;
procedure TMainForm.OnClear;
begin
  Memo.Clear;
end;
```

➤ *Listing 8*

➤ *Below: Figure 5*



My dog has fleas
Hello Mr. Automation!

Hello Mr. Automation!

My dog has fleas
Hello Mr. Automation!