CHAPTER 6

The ADO Event Model and Asynchronous Processing

ADO 2.0 INTRODUCED a comprehensive set of events on the Connection and Recordset objects. One of the most common questions developers ask when learning about ADO events is—interesting, but why would I use them?

Every VB programmer is familiar with events because the process of creating user interfaces in VB involves placing controls, which define their own set of events, on a form and writing code in event procedures. The VB code window lists each control in the Objects dropdown list and the developer can select an event in the Procedures dropdown list. Code can then be written that will execute when these events fire.

VB5 introduced the ability to receive events from standard *COM* classes, as well as controls. When an object variable is declared using the WithEvents keyword, the object variable name is treated like a control name. The variable name appears in the Objects dropdown list, and its events are listed in the Procedures dropdown list. They can be coded just like a control's events.

A control fires events to allow its client to respond to something that has happened. The client can then tailor its behavior (and therefore the application or its user interface) appropriately. If controls did not raise events they would be far less useful. Imagine writing VB programs using a CommandButton that did not raise a Click event!

You should think about the events raised by the ADO Connection and Recordset objects in the same way as you think about the events raised by a control and exploit them for the same reasons.

Many of ADO's events come in pairs. For example, one of the most important things a Connection object does is to connect to a data source. ADO provides two events associated with this operation. The cn_WillConnect event fires before a Connection object starts the process of connecting to a data source, and the cn_ConnectComplete event fires once the connection process is completed. Event pairs give you a substantial degree of control over key ADO tasks.

ADO allows certain processes to be started asynchronously, which means a client program can continue working even while the process of connecting or fetching data is taking place. Events allow code to receive notifications when a task is complete, and it's reasonably easy to see how they can be useful in asynchronous tasks. However, it would be unfortunate to equate event processing to asynchronous operations. If you did, you would miss out on the very high level of control they give you over any ADO program, asynchronous or not. My aim in this chapter is to set out some broader scenery in which the role of events can be evaluated and understood. For sure, we'll look at asynchronous processing, but only after considering the full picture.

Connection Events

To demonstrate what can be achieved by using events, let's look at how to add an ADO Connection as an event monitor to an ADO application. The monitor can be used to diagnose ADO errors in a compiled application, or even to change the connection string so that the monitored application connects to a test database. The monitor itself will be written as a DLL that exploits ADO Connection events. The beauty of this technique is that

1. To use the monitor requires adding only six lines of code to the application being monitored.

2. When the monitor isn't being used, there is no performance overhead in the application, and no special coding is required (apart from the six lines of code).

3. You can add and remove monitors to a compiled program without making any coding changes. You can switch between multiple monitors, and the monitor doesn't even need to have been written when the "client" application is compiled and distributed.

4. The monitor will work with any ADO application that uses explicit Connection objects and includes the six lines of code or their equivalent.

I promise not to let the details of this sample program get in the way of finding out about events. It so happens that it's an easy way of creating a dramatic example of what they can be used for. First, however, let's take a look at how a Connection object's events are organized.

Table 6-1. Connection Events

PAIRED EVENTS	
Before Operation	***After Operation***
WillConnect	ConnectComplete
WillExecute	ExecuteComplete
UNPAIRED EVENTS	
BeginTransComplete	
CommitTransComplete	
RollbackTransComplete	
Disconnect	
InfoMessage	

I won't be describing each event formally. Instead, code samples and associated narrative will contain all the details required.

Each event has its own set of arguments, but there is some common ground that makes understanding events a little easier. Every event has an argument called adStatus whose value is taken from the EventStatusEnum enumeration when the event fires. It has these values defined:

adStatusOK
adStatusErrorsOccurred
adStatusCantDeny
adStatusCancel
adStatusUnwantedEvent

adStatusOK means that the operation is moving along fine, while adStatusErrorsOccurred means the opposite. Nearly every event has a pError argument, which is set to an ADO Error object when `adStatus` equals `adStatusErrorsOccurred`. You can use this Error object to find out more about any error that occurs.

In many cases, you can use the event to cancel a pending operation by setting `adStatus` to `adStatusCancel` inside the event procedure of a "Will..." event. However, if `adStatus` equals `adStatusCantDeny` then the operation can't be cancelled (in a "…Complete" event, adStatusCantDeny means that the operation has already been cancelled). If you cancel an event in this way, an error will be raised in the procedure that triggered the event, and therefore, adequate error handling will be required.

Setting `adStatus` to `adStatusUnwantedEvent` allows you to instruct ADO to stop firing that particular event. There is a cost associated with raising events,[1] so blocking unwanted ones has some performance benefit.[2]

In addition to adStatus and pError arguments, the events provide references to Connection, Command, and Recordset objects where appropriate, as well as some event-specific arguments.

cn_WillConnect
cn_Connect-Complete
cn_Disconnect
cn_InfoMessage

Let's begin by looking at the WillConnect, ConnectComplete, Disconnect, and InfoMessage events. The first three should be self-explanatory. InfoMessage is raised when an ADO Error object is created, and when the error is not so severe as to cause a run-time error in VB.

In my sample applications, all ADO event monitoring is performed in an ActiveX DLL called ADOLogger, which contains a Public class called ADOConnection (and therefore its ***ProgID*** is ADOLogger.ADOConnection).

You'll see that the client program's code uses a standard VB global variable called Command. This is nothing to do with ADO's Command objects. The Command variable holds any command-line argument that is provided when the application is executed. For example, if the client was compiled as `c  myapp.e e`, then running it as `c  myapp.e e ADOLogger.ADOConnection` would allocate the Logger's ProgID to the Command variable, which could then be used in the CreateObject call. During development, you can simulate using a command-line argument in the VB IDE via the Make tab of the Project Properties dialog.

1. You are unlikely to notice the cost of Connection object events. However, the cost of Recordset events becomes significant when iterating through a client-side Recordset. Setting an event as unwanted can easily reduce this overhead by half.

2. Readers familiar with COM may be aware that events in VB are always handled using early binding. While this is more efficient than the late binding that results from declaring a variable "As Object," it's far less efficient than the vTable binding that can be used for methods and properties when variables are defined as a specific class.

Here's how the client application communicates with the Logger:

```
Private cn As Connection
Private oLog As Ob"ect
```

the client progra declares a ariable to reference the ogger

```
Private Sub Form_Load &
Set cn = New Connection
If Command  , "" T en
   Set oLog = CreateOb"ect Command&
   oLog.Monitor cn
   End If
cn.Open "File Name=c  Muc ADO.udl"
End Sub
```

if a co and line argu ent is supplied it s treated as a rog for a ogger ob ect

the client passes a reference to its Connection ob ect to the ogger

```
Private Sub Form_Unload Cancel As Integer&
cn.Close
Set cn = Not ing
If Command  , "" T en Set oLog = Not ing
End Sub
```

the client releases the ogger hen it releases the Connection ob ect to ensure that the Connec tion ob ect ter inates

From this point on, the Logger can monitor all activity on the cn Connection through its events without any additional Logger-related code in the client program. The Logger is only used when the built-in VB global variable `Command` indicates that a command-line argument was provided when the application was executed. By using `CreateOb"ect`, the client doesn't have any dependencies on the Logger compiled into it other than the expectation that the Logger (if used) will support the `Monitor` method. This, of course, means that I could write different Loggers for different tasks and connect to any such Logger at run time.

Now let's look at the Logger. Remember that this is written as an ActiveX DLL, which is compiled separately from the main EXE (although you could incorporate the Logger code into the main client EXE if you wanted to). Here's the code for the ADOLogger.ADOConnection class (the event argument lists are shown in gray to make it easier to identify the actual code):

```
Private 1it Events cn As Connection
Private dOpTime As Double
```

cn ariable declared ith ents

```
Public Sub Monitor cnToMonitor As Connection&
  Set cn = cnToMonitor
End Sub
```

this is the onitor ethod called by the client hich assigns the Connection ariable

```
Private Sub cn_InfoMessage  _
        By3al pError As ADODB.Error, _
        adStatus As ADODB.EventStatusEnum, _
        By3al pConnection As ADODB.Connection&
  MonitorEvent adStatus, "Info Message  " 5 pError.Description
End Sub


Private Sub cn_1illConnect  _
        ConnectionString As String, _
        UserID As String, _
        Password As String, _
        Options As Long, _
        adStatus As ADODB.EventStatusEnum, _
        By3al pConnection As ADODB.Connection&
  MonitorEvent adStatus, "Connecting to  " 5 ConnectionString
  dOpTime = Timer
End Sub

Private Sub cn_ConnectComplete  _
        By3al pError As ADODB.Error, _
        adStatus As ADODB.EventStatusEnum, _
        By3al pConnection As ADODB.Connection&
  dOpTime = Timer 6 dOpTime
  If adStatus = adStatusErrorsOccurred T en
    MonitorEvent adStatus, pError.Description
  Else
    MonitorEvent adStatus, "Connection succeeded in " 5 _
         Format dOpTime, "7.77"& 5 _
         " seconds to " 5 pConnection.ConnectionString
  End If
End Sub


Private Sub cn_Disconnect  _
        adStatus As ADODB.EventStatusEnum, _
        By3al pConnection As ADODB.Connection&
    MonitorEvent adStatus, "Disconnected"
End Sub
```

This code listing doesn't show the code for the `MonitorEvent` subroutine, which simply converts `adStatus` to a printable form, and prints it to the Debug window in the VB IDE. When using the compiled version of the Logger, MonitorEvent writes data to the NT Event Log (or a file on Windows 9.x).

When the client application is run with no command-line argument, the Logger is never started, no events get processed, and nothing is printed. However, if the client application is run with the Logger's ProgID in the command line, the following is printed:

```
08            Connecting to  File Name=c  Muc ADO.udl
08            Connection succeeded in 7.9: seconds to
Provider=S;LOLEDB.9<Integrated Security=SSPI<Persist Security
Info=False<Initial Catalog=Muc ADO<Data Source=POLECAT<Use Procedure for
Prepare=9<Auto Translate=True<Packet Si>e= 7 :<1orkstation ID=1EASEL
08            Disconnected
```

If I change the connection string to use an ODBC-based connection, I'll see an example of the InfoMessage event firing, as in the following printout:

```
08            Connecting to  DSN=Muc ADO
08            Connection succeeded in 7.AA seconds to
Provider=MSDAS;L.9<E tended Properties="DSN=Muc ADO<UID=<APP=3isual
Basic<1SID=1EASEL<DATABASE=Muc ADO<Trusted_Connection=Yes"
08            Info Message  CMicrosoftDCODBC S;L Server DriverDCS;L
ServerDC anged database conte t to 'Muc ADO'.
08            Disconnected
```

If you take a look at some of the arguments to the various events in the Logger code, you'll get an idea of the flexibility you can achieve when handling events. For example, let's assume that the client uses a hard-coded connection string, but you want to divert the client to a test database during monitoring.[3] Inserting the line

```
ConnectionString = "File Name=c  Muc AdoFet.udl"
```

into the `1illConnect event` will cause the `ConnectionString` argument to be changed, and will result in a different data source connection being created:

```
08            Connecting to  DSN=Muc ADO
08            Connection succeeded in 7.GH seconds to
Provider=Microsoft.Fet.OLEDB. .7<User ID=Admin<Data
Source=D  ADOBook Data muc ado.mdb<  etc&
08            Disconnected
```

3. This example demonstrates the potential security loophole that can result from using a Logger like this. You'll probably want a real-life Logger to implement appropriate security measures.

cn_BeginTrans-Complete
cn_CommitTrans-Complete
cn_RollbackTrans-Complete

The Connection object also supports events for transaction processing commands. There are no "Will…" events in this case, just "…Complete" events. Incorrect transaction processing can result in surprising errors, and analyzing what is really going on with transactions can be complex. Long-running transactions can cause serious concurrency problems in multiuser applications. In such situations, an activity logging mechanism can be invaluable.

Here's the transaction monitoring code in the Logger:

```
Private Sub cn_BeginTransComplete  _
        By3al TransactionLevel As Long, _
        By3al pError As ADODB.Error, _
        adStatus As ADODB.EventStatusEnum, _
        By3al pConnection As ADODB.Connection&
Dim sTI As String                                   print the ti e the transaction
                                                    started  and the isolation
  If adStatus = adStatusO8 T en                     le el  a ailable through the
    dOpTime = Timer                                 pConnection argu ent
    sTI = "Beginning TI  Isolation Level " 5 _
          pConnection.IsolationLevel 5 "& at " 5 _
                Format Now, "   mm ss"&
  Else
    sTI = "Begin Transaction Error " 5 pError.Description
  End If
  MonitorEvent adStatus, sTI
End Sub

Private Sub cn_CommitTransComplete  _
        By3al pError As ADODB.Error, _
        adStatus As ADODB.EventStatusEnum, _
        By3al pConnection As ADODB.Connection&
Dim sTI As String
  If adStatus = adStatusO8 T en
    dOpTime = Timer 6 dOpTime
    sTI = "Committing TI  " 5 Format dOpTime, "7.77"& 5 " seconds&"
  Else
    sTI = "Commit Transaction Error " 5 pError.Description
  End If
  MonitorEvent adStatus, sTIEnd Sub

Private Sub cn_RollbackTransComplete  _
        By3al pError As ADODB.Error, _
        adStatus As ADODB.EventStatusEnum, _
        By3al pConnection As ADODB.Connection&
```

```
Dim sTI As String
  If adStatus = adStatusO8 T en
    dOpTime = Timer 6 dOpTime
    sTI = "Rolling back TI  " 5 Format dOpTime, "7.77"& 5 " seconds&"
  Else
    sTI = "Rollback Transaction Error " 5 pError.Description
  End If
  MonitorEvent adStatus, sTI
End Sub
```

Note that the `cn_BeginTransComplete` event has an argument called `Transaction6 Level`. It tells you the nesting level of the new transaction. This is only really useful when you are using a Provider that supports nested transactions, such as the Jet Provider.

The following client code

```
cn.BeginTrans
cn.E ecute "update Parts set description = 'bit part' " 5 _
                  "w ere part = 'BOY'"
cn.RollbackTrans
```

generates

```
  O8            Beginning TI  Isolation Level  7 :& at 9K AG  7
  O8            Rolling back TI  7.7  seconds&
```

The transaction events fire only when the relevant ADO methods are explicitly called. This means

- Statements not bracketed by explicit ADO Connection transaction methods will be updated inside an implicit transaction and no event traffic will be generated.
- None of the events will fire when transactions are being managed by MTS or COM+.[4]

cn_WillExecute
cn_Execute-Complete

Let's finish this section by looking at the WillExecute and ExecuteComplete events. Here's an example from the Logger DLL:

4. Although these products have their own event mechanisms for logging transactional activity.

```
Private Sub cn_1illE ecute  _
        Source As String, _
        CursorType As ADODB.CursorTypeEnum, _
        LockType As ADODB.LockTypeEnum, _
        Options As Long, _
        adStatus As ADODB.EventStatusEnum, _
        By3al pCommand As ADODB.Command, _
        By3al pRecordset As ADODB.Recordset, _
        By3al pConnection As ADODB.Connection&
Dim sOb"ects As String
If pCommand Is Not ing T en sOb"ects = "No Command Ob"ect "
If pRecordset Is Not ing T en sOb"ects = _
         sOb"ects 5 "No Recordset Ob"ect "
MonitorEvent adStatus, sOb"ects 5 vbCrLf 5 vbTab 5 _
         " " 5 CursorType 5 "," 5 LockType 5 "& " 5 Source
End Sub
```

It's interesting to note that although this event belongs to a Connection object, it will fire when rs.Open or cd.Execute is called, if cn is the active connection. Therefore, the following client code

```
Dim rs As New Recordset
Dim cd As New Command
cn.E ecute "SELECT * FROM Parts"
cd.CommandTe t = "SELECT * FROM Scenes"
cd.ActiveConnection = cn
cd.E ecute
rs.Open "SELECT * FROM SceneContents", cn
```

will generate

```
O8          No Command Ob"ect No Recordset Ob"ect
                 69,69& SELECT * FROM Parts
O8          No Recordset Ob"ect
                 69,69& SELECT * FROM Scenes
O8          No Command Ob"ect
                 7,69& SELECT * FROM SceneContents
```

This sample illustrates a number of points. The `pCommand` and `pRecordset` arguments will only contain objects if you supply them. This is a "Will…" event, so unless you are calling rs.Open, the Recordset is created during the Execute operation and isn't available when the "Will…" event fires. This has an understandable but unfortunate consequence for the `CursorType` and

LockType arguments of the WillExecute event. CursorType and LockType are properties of a Recordset object. If there is no Recordset object, it doesn't make much sense to set these properties. This sad fact rules out the chance of exploiting WillExecute to change Recordset properties when cn.Execute or cd.Execute is used.[5]

You can however, set the cn.CursorLocation property to force a client-side (and therefore static) cursor. And of course, there is nothing to stop you from changing the Source argument if a worthwhile reason comes to mind.

After the execution completes, the cn_ExecuteComplete event fires. Here's an example from the Logger:

```
Private Sub cn_E ecuteComplete  _
        By3al RecordsAffected As Long, _
        By3al pError As ADODB.Error, _
        adStatus As ADODB.EventStatusEnum, _
        By3al pCommand As ADODB.Command, _
        By3al pRecordset As ADODB.Recordset, _
        By3al pConnection As ADODB.Connection&
Dim sOb"ects As String

If pCommand Is Not ing T en sOb"ects = "No Command Ob"ect "
If pRecordset Is Not ing T en _
   sOb"ects = sOb"ects 5 "No Recordset Ob"ect "

MonitorEvent adStatus, " " 5 RecordsAffected 5 "&" 5 sOb"ects
End Sub
```

And here's the printout that results from this event, using our existing client code, and with all other event logging suppressed:

```
08              69&
08              69&
08              69&
```

This is not very exciting. It does show that a Command object and a Recordset object have been created during each execution process, and it also shows that RecordsAffected returns –9 for a SELECT statement.

To seek out some more exciting output, I used the following client code against the Jet Provider:

5. You probably don't need to know this, but when there is no Recordset object, ADO passes the same memory address for both the CursorType and LockType arguments, so setting one automatically sets the other. This is the kind of worthless fact that gets you noticed at parties.

```
Dim rs As New Recordset
Dim cd As New Command
cn.E ecute "UPDATE Parts SET description = Null " 5 _
           "1MERE part = 'BOY'"    , , adE ecuteNoRecords
cd.CommandTe t = "SELECT * FROM Scenes"
cd.ActiveConnection = cn
cd.E ecute
rs.Open "Parts", cn, , , adCmdTableDirect
```

which resulted in

```
O8                    9&No Recordset Ob"ect
O8                    7&
Errors Occurred       7&No Command Ob"ect
```

Without the `adE ecuteNoRecords` argument, `cn.E ecu`te would have generated a closed Recordset. Using `adCmdTableDirect` is the only way of not creating a Command object. It has the interesting effect of reporting an error via adStatus, even though no Error object is created and the client code proceeds perfectly.

Recordset Events

Recordset objects also have a comprehensive set of events that can be received by a variable declared using WithEvents,[6] as shown in the following table.

Table 6-2. Recordset Events

PAIRED EVENTS	
Before Operation	***After Operation***
WillChangeField	FieldChangeComplete
WillChangeRecord	RecordChangeComplete
WillChangeRecordset	RecordsetChangeComplete
WillMove	MoveComplete
UNPAIRED EVENTS	
EndOfRecordset	
FetchProgress	
FetchComplete	

6. The ADO Data Control provides a very similar set of events for developers who use it.

cn_WillChangeField
cn_FieldChange-Complete
cn_WillChange-Record
cn_RecordChange-Complete
cn_WillChange-Recordset
cn_Recordset-ChangeComplete
cn_WillMove
cn_MoveComplete

The paired events are almost, but not quite, self-explanatory. FetchProgress and FetchComplete are only relevant to asynchronous processing, and they will be discussed in a later section. EndOfRecordset is an unusual event—we'll take a look at it shortly.

First however, let's look at the paired events. These allow you to respond to many standard Recordset operations is a very fine-grained way. Apart from writing logging or monitoring applications, one of the primary reasons for using these events is to separate navigation and user interaction from the underlying processing, validation, and business logic associated with a particular Recordset.[7]

As an example, consider the following VB Class. It returns a Recordset based on the Parts table and implements validation code so that only an administrator can delete records or change the part name, but any user can change a part description.

```
'********** CODE FOR CLASS PARTS *****************
Public 1it Events rs As Recordset
Private msUser As String

Public Sub GetData cn As Connection&
Set rs = New Recordset
rs.CursorLocation = adUseClient
rs.LockType = adLockOptimistic
rs.Open "SELECT * FROM parts", cn
msUser = rs.ActiveConnection.Properties "User ID"&
End Sub

Private Sub rs_1illC angeField  _
        By3al cFields As Long, _
        By3al Fields As 3ariant, _
        adStatus As ADODB.EventStatusEnum, _
        By3al pRecordset As ADODB.Recordset&
On Error GoTo ErrM
Dim vField As 3ariant                    <-- don t do any chec s for
If msUser = "sa" T en E it Sub               the ad inistrator' she
For Eac  vField In Fields                    can do anything she li es
  If vField.Name = "part" T en adStatus = adStatus
Ne t
ErrM
End Sub
```

7. Visual programming approaches (Data Control, Data Environment, Data Repeater, DHTML) work by taking care of the navigational and user interaction aspects of a Recordset, leaving you with the event model to control functionality.

```
Private Sub rs_1illC angeRecord  _
        By3al adReason As ADODB.EventReasonEnum, _
        By3al cRecords As Long, _
        adStatus As ADODB.EventStatusEnum, _
        By3al pRecordset As ADODB.Recordset&
If msUser = "sa" T en E it Sub          <-- don t do any chec s for the ad inistrator' she can do anything she li es
If adReason = adRsnDelete T en adStatus = adStatus
End Sub
```

In this class, `GetData` must be called to create a Recordset, which is then available as a property on the object. By implementing Recordset events, the class effectively provides a Recordset with extended functionality, customized to serve the needs of a particular Recordset (in this case, cancelling certain operations unless the user name is "sa").

Using this class with the following code works fine:

```
On Error GoTo ErrM
Dim cn As New Connection
Dim oPart As New Parts
cn.Open "DSN=Muc ADO", "user", "user"
1it  oPart
  .GetData cn
  .rs.Filter = "part = 'BOY'"
  '.rs!part = "GIRL"  <-- this line is co  ented out
  .rs!Description = "a young male"
  .rs.Update
End 1it
E it Sub
ErrM
Print Err.Description
```

However, restoring the line that is commented out results in

```
Operation was cancelled.
```

unless the user is changed to "sa". Similar behavior results if a Delete is attempted.

Now that you have seen an example of the paired events in operation, let me add a bit more detail. You may be thinking that record-level events fire for every Field-level event and record-level event, and that Recordset-level events fire for just about every operation. This isn't how it works. The Field-level events fire only when you perform a Field object operation, such as setting a Value property. The record-level events fire only for those operations

relevant to a whole record, while Recordset-level events fire only for operations that affect the entire Recordset. With the exception of the Field-level events, each paired Recordset-level event carries an adReason argument, which contains a value from the EventReasonEnum enumeration. This provides additional information about which operation caused an event to fire. The following table lists the operations that cause events to fire at a particular level, and where appropriate, gives the reason code associated with the operation.

Table 6-3. Events Fired by Different Recordset Operations

ADO OPERATION	REASON	FIELD-LEVEL EVENTS	RECORD-LEVEL EVENTS	RECORDSET-LEVEL EVENTS
fd.Value	adRsnFirstChange	Yes	Yes[8]	
rs.Update, rs.UpdateBatch	adRsnUpdate	Yes[9]	Yes[10]	
rs.AddNew	adRsnAddNew		Yes	
rs.Delete	adRsnDelete		Yes	
rs.CancelUpdate, rs.CancelBatch	adUndoUpdate, adUndoAddNew, adUndoDelete		Yes	
rs.Requery	adRsnRequery			Yes
rs.Resync	adRsnReSynch			Yes
rs.Close	adRsnClose			Yes

The EventReasonEnum also contains values used to indicate the type of Move operation that triggered a Move event. Any operation that changes the current cursor position can trigger a Move event, notably including rs.Open and rs.Filter.

The fact that the first edit operation on the current record raises a record-level event with adRsnFirstChange as the reason code can be very helpful. For example, consider the situation when you have a Clone of a Recordset, and the Clone is pointing at a different record than the original Recordset was pointing at. When the original Recordset updates a Field, the Field-level

8. The first time a field is updated after a Move operation, the record-level events and the Field-level events will fire.

9.When Update is called with field name and value arrays, the Field-level events and the record-level events fire.

10.Update doesn't trigger a record-level event when in batch update mode.

events will fire on both the original Recordset and the Clone. However, the Field-level events don't tell you which record has just been updated, only which Fields are affected. This doesn't matter for the original Recordset, because it knows which record it has updated. However, the Clone doesn't have this knowledge. Fortunately, when the record-level events fire on the Clone (for the first update only), the pRecordSet argument has a filter applied that identifies the current record.

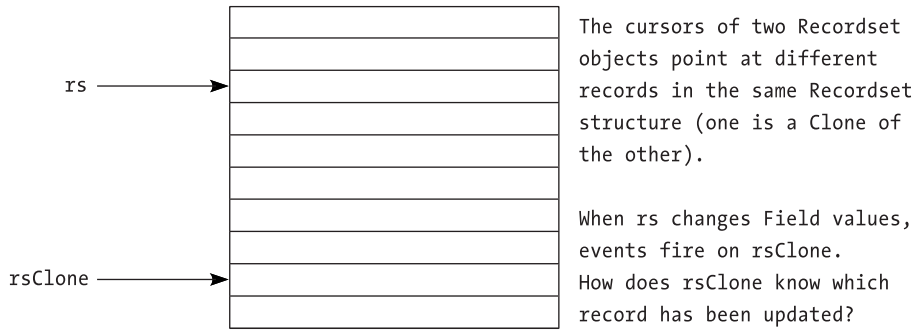

Figure 6-1. Events can help a Recordset identify which record a Clone has updated.

The following code demonstrates this. It assumes that `rsClone` has been declared using WithEvents as a module-level variable.

```
Dim rs As New Recordset
rs.CursorLocation = adUseClient
rs.Open "SELECT * FROM Parts", cn _
      , adOpenStatic, adLockOptimistic
Set rsClone = rs.Clone
rs.Filter = "part = 'BOY'"
rsClone.MoveLast
Debug.Print rsClone!part
rs!part = "GIRL"
rs!Description = "a young female"
rs.Update

Debug.Print rsClone!part
```

print the part that the Clone is currently pointing to

update the record identified by the filter

print the part that the Clone is currently pointing to

```
Private Sub rsClone_1illC angeRecord  _
        By3al adReason As ADODB.EventReasonEnum, _
        By3al cRecords As Long, _
        adStatus As ADODB.EventStatusEnum, _
        By3al pRecordset As ADODB.Recordset&
Debug.Print "E3ENT SAYS", pRecordset!part 5 _
               " " 5 adReason 5 "&"
End Sub
```

print the part that the e ent filter identifies as the changed record

It prints

```
RMETT BUTLER
E3ENT SAYS    BOY     99&          <-------  (( ) ad sn irstChange
E3ENT SAYS    GIRL    G&           <-------  ,   ) ad sn-pdate
RMETT BUTLER
```

Stepping through this in your mind will identify that `rsClone_1ill6 C angeRecord` is called twice—once when the record is first changed, and once when the `rs.Update` takes place. The filter is applied only during the event procedure, and it allows the Clone to know which record has changed, even when it's currently pointing at a different record.

cn_EndOfRecordset

Let's close this section by looking at the EndOfRecordset event. This ADO event allows you to populate a fabricated Recordset incrementally. Assume that you have a potentially large source of data that you want to present as a Recordset. A user of your Recordset might only want to use the first few records or may want to see thousands. It might take you some time to populate a Recordset with thousands of records. The EndOfRecordset event allows you to return with just a handful of records initially. When the user attempts to read past the last record, it allows you to add the next handful into the Recordset when the EndOfRecordset event fires, potentially ad infinitum.

For example, consider the following Recordset in which each record contains a random number (assuming `rsRandom` has been declared using WithEvents as a module-level variable).

```
Private Sub Form_Load &
Set rsRandom = New Recordset
rsRandom.Fields.Append "Ne t", adSmallInt
rsRandom.Open
End Sub
```

```
Private Sub rsRandom_EndOfRecordset  _
        fMoreData As Boolean, _
        adStatus As ADODB.EventStatusEnum, _
        By3al pRecordset As ADODB.Recordset&
pRecordset.AddNew "Ne t", CInt Rnd * 977&
pRecordset.AddNew "Ne t", CInt Rnd * 977&
fMoreData = True
End Sub

Private Sub Command9_Click &
Print rsRandom!Ne t, rsRandom.RecordCount
rsRandom.MoveNe t
End Sub
```

.etting fMoreData to True tells that ore data can no be read1 ea ing it as False eans that the ecordset has genuinely reached its end1

Repeatedly hitting the Command1 button yields these results:

```
K9          7
AG          H
AP
H
G7          :
KK          :
9           P
K:          P
P9          97
K9          97
A           9H
```

In this case, you don't really know how many records are going to be required, and you don't want to create more than necessary. The EndOf-Recordset event can help you in such situations.

Strategies for Using Events

Unless you are using asynchronous operations (see next section) it's unlikely that you'll make much use of ADO events in small applications. There is little need in such applications to take on board the extra discipline required, because the benefits are not sufficient. You may start to think differently