The Report Engine Automation Server

Seagate introduced the Report Engine Automation Server (known simply as the "Automation Server") with Crystal Reports 6. At the time, this integration method was innovative, particularly for Visual Basic programmers. Instead of the "drop-the-object-on-the-form" approach of the ActiveX control, the Automation Server was the first Crystal Reports programming interface to comply with Microsoft's then-new Component Object Model, or COM.

Shortly after Crystal Reports 6 was released, however, Seagate introduced the Report Designer component (discussed in Chapter 23). This programming interface expanded on the COM model, providing not only a Runtime Automation Server of its own, but also a design-time report designer that works right inside the Visual Basic development environment. With Crystal Reports 8, Seagate has stated its intention to provide support only for newer reporting features with the RDC—the Automation Server will no longer be updated (and actually hasn't been since Crystal Reports Version 6). As such, this chapter appears in the book as a reference for those who will be modifying existing Automation Server applications that, for one reason or another, aren't candidates for conversion to the RDC.

Overview of the Automation Server

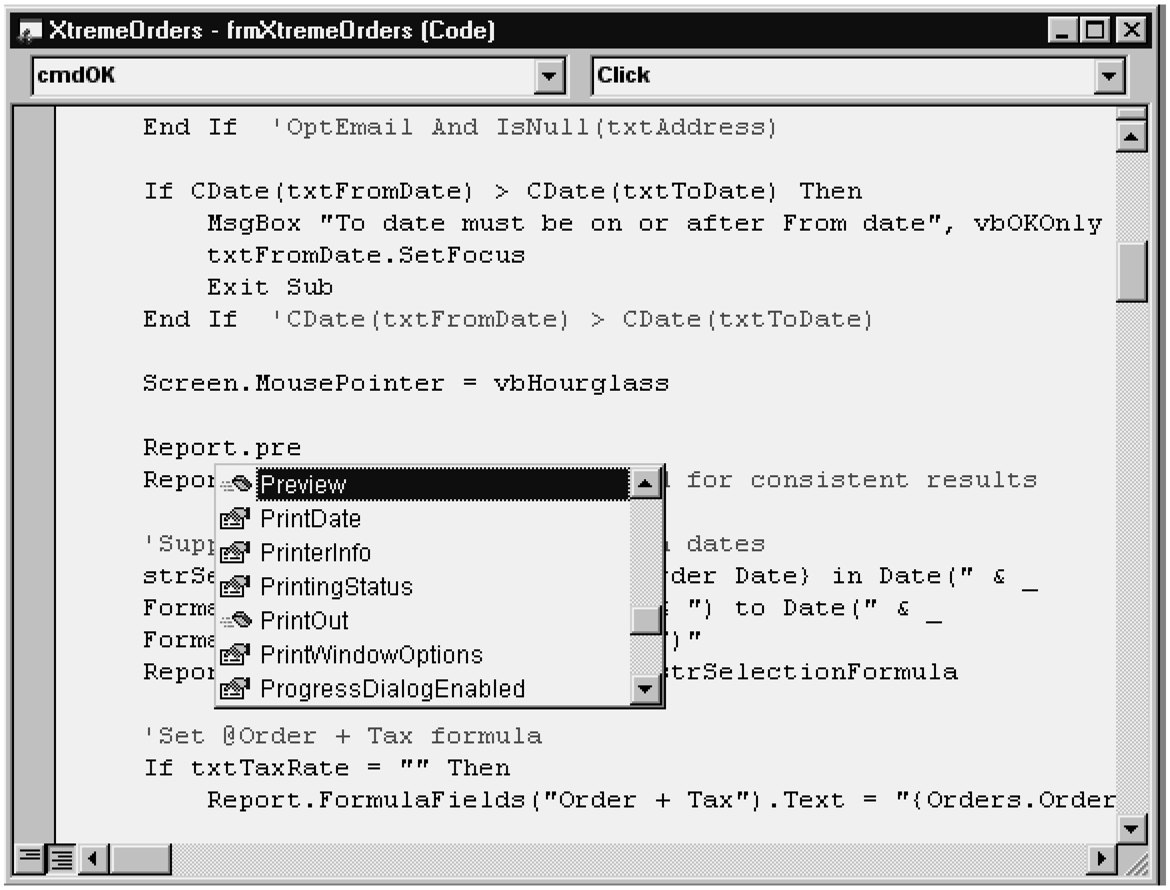
The Automation Server conforms to Microsoft's COM. As such, it can be used in any COM-compatible development environment that supports a COM Automation Server (formerly known as an "OLE server"), such as Visual Basic, Visual C++, Delphi, and Microsoft Visual InterDev for Web page development. While an in-depth discussion of COM is better left to Microsoft documentation, you can consider COM to be a set of standards that allows one software environment to utilize the functions of another software environment by way of objects, properties, methods, and events.

COM development falls under the category of object-oriented programming (OOP). An *object hierarchy* is exposed by the COM automation server. This hierarchy begins with high-level objects that contain lower levels of objects. These second-level objects can contain lower levels of objects, and so on. If it sounds like the levels of objects can get "deep," they sometimes do. Some objects in the Automation Server are four or five layers deep in the object hierarchy.

Objects can contain not only additional single objects below them, but also collections of objects. A *collection* is a group of related objects contained inside the parent object. For example, a Report object will contain a collection of Section objects (an object for the report header, page header, and so on). Each of these Section objects itself contains its own collection of Report objects, such as fields, text objects, formulas, lines, boxes, and so on. If the details section of a report has ten database fields, four formulas, and a line-drawing, the Details Section object will contain an object collection consisting of 15 *members*—one member for each of the Report objects in the section. And, if a report simply contains the five default sections that are created with a new report, the report's Sections collection will contain five members—one for each section.

The highest level in the object hierarchy is the Application object—everything eventually "trickles down" from it. Although you do need to declare the Application object in your VB code, you will rarely (if ever) refer to it again, or set any properties for it, once it's declared. As a general rule, the highest level of object you'll regularly be concerned with is the *Report* object, created by executing the Application object's OpenReport method. You will then manipulate many of the Report object's properties (such as the RecordSelectionFormula property), as well as work with many objects below the Report object (such as report sections, as just discussed).

Because the Automation Server is a tightly integrated COM component, you will benefit from Visual Basic's *automatic code completion* when you use it. When you begin to enter code into the VB code window for the Report object, the properties, methods, and built-in constant values become available to you from automatically appearing drop-down lists. If you've used Visual Basic 5 or 6, you're probably already familiar with this feature. You'll enjoy the benefit it provides with the Automation Server, as well.



```
    End If  'OptEmail And IsNull(txtAddress)

    If CDate(txtFromDate) > CDate(txtToDate) Then
        MsgBox "To date must be on or after From date", vbOKOnly
        txtFromDate.SetFocus
        Exit Sub
    End If  'CDate(txtFromDate) > CDate(txtToDate)

    Screen.MousePointer = vbHourglass

    Report.pre
    Repor                           l for consistent results

    'Supp                           i dates
    strSe                           rder Date) in Date(" & _
    Forma                           ") to Date(" & _
    Forma                           )"
    Repor                           trSelectionFormula

    'Set @Order + Tax formula
    If txtTaxRate = "" Then
        Report.FormulaFields("Order + Tax").Text = "{Orders.Order
```

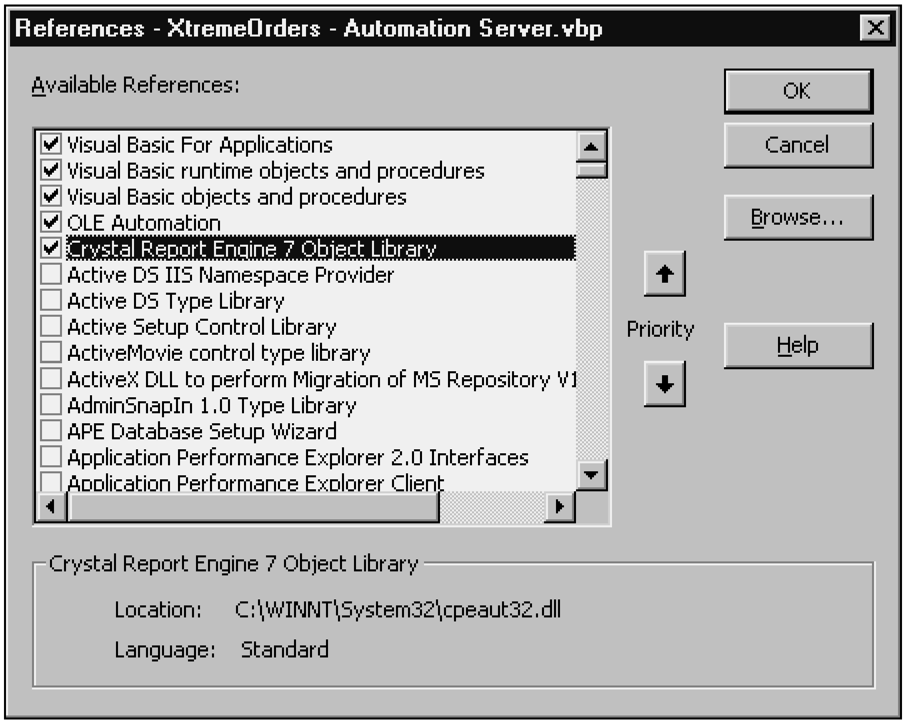
Caution: Unlike the RDC (discussed in Chapter 23), the Automation Server automatically completes code only to one level in the object hierarchy. When you type the period to go to the next level, you'll just hear a beep, and nothing else will display in the code window. If you want to proceed further down the hierarchy, you need to make sure that you type the correct syntax for the deeper objects and properties. Check the Object Browser (discussed later in this chapter) or online Help for information on correct syntax.

Adding the Automation Server to Your Project

The first step in using the Automation Server is to add it to your project. After installing Crystal Reports 8 Developer Edition, the Automation Server should be properly registered on your system and ready for use. Unlike previous versions, Crystal Reports 8 only includes the 32-bit version of the Automation Server, contained in a file named CPEAUT32.DLL.

To add the Automation Server to your project, follow these steps:

1. Start Visual Basic and create a new project or open the existing project that you wish to use with the Automation Server.
2. Choose Project | References from the pull-down menus. The References dialog box will appear.
3. Select the Crystal Report Engine 8 Object Library check box. If you don't see this option, click the Browse button and locate the Automation Server .DLL file (CPEAUT32.DLL) in the Windows System directory.



- [✓] Visual Basic For Applications
- [✓] Visual Basic runtime objects and procedures
- [✓] Visual Basic objects and procedures
- [✓] OLE Automation
- [✓] Crystal Report Engine 7 Object Library
- [ ] Active DS IIS Namespace Provider
- [ ] Active DS Type Library
- [ ] Active Setup Control Library
- [ ] ActiveMovie control type library
- [ ] ActiveX DLL to perform Migration of MS Repository V1
- [ ] AdminSnapIn 1.0 Type Library
- [ ] APE Database Setup Wizard
- [ ] Application Performance Explorer 2.0 Interfaces
- [ ] Application Performance Explorer Client

Crystal Report Engine 7 Object Library

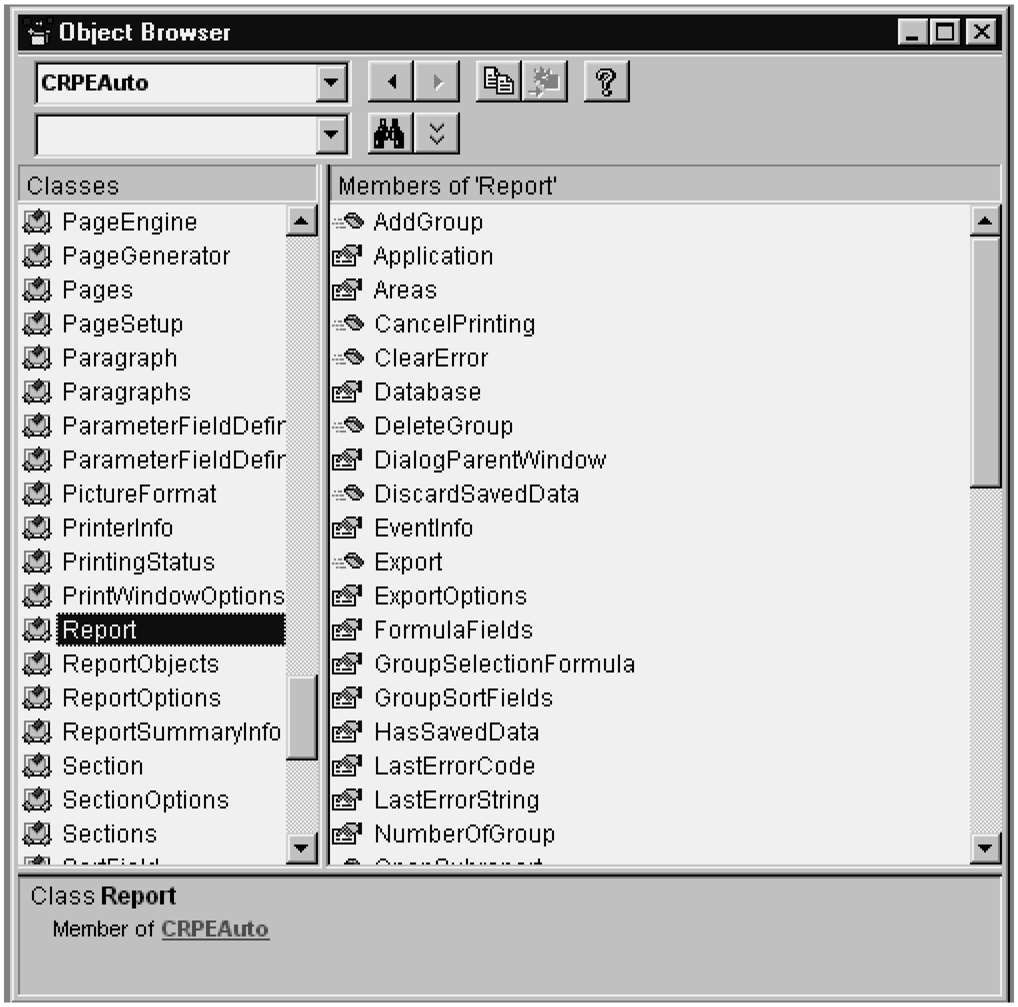
Location: C:\WINNT\System32\cpeaut32.dll

Language: Standard

Note: These instructions apply to Visual Basic 5 and 6. Check Microsoft documentation for information on adding COM automation servers to a VB 4 project.

When you add the Automation Server to your project, there are two immediate ways of viewing the various objects, collections, properties, methods, and events that are exposed. The first is simply by using the online Developer's Help provided with Crystal Reports. This is available by navigating to the DEVELOPR.HLP file, located in *Crystal Reports program directory*\Developer\Help.

You can also use Visual Basic's *Object Browser,* which can be viewed in Visual Basic any time your program isn't running. Press F2 or choose View | Object Browser from the pull-down menus. Then, choose CRPEAuto in the first drop-down list to see the object library exposed by the Automation Server.



Declaring the Application and Report Objects

After you actually add the Automation Server library to your project, the first step is to declare Application and Report objects. These are the upper-level objects in the object hierarchy and must be declared for every project. There generally should be only one Application object per VB project. If you will be manipulating more than one report at a time, you can declare multiple Report objects within the Application object (or, just reuse a single Report object if you are using multiple reports but only need to use one particular report at a time).

Both objects are initially declared with the Dim statement in an appropriate Declarations section of a form or Sub Main. In the Xtreme Orders sample application, these are declared in the Declarations section of the form:

```
Dim Application As CRPEAuto.Application
Dim Report As CRPEAuto.Report
```

Note that the object types have been *fully qualified.* That is, they are not declared with the simple generic "Object" type. Instead, the Automation Server Application and Report object types are used. To take this a step further, the Automation Server CRPEAuto library name is added to the object types, separated by a period, to explicitly indicate that they are to refer to the Crystal Reports Automation Server. Using this specific class ID avoids confusion with other objects of the same type that may also be added to the VB project (such as the DAO library or another library with identical object types). This full qualification facilitates Visual Basic's "early binding," which improves application performance, as well as allowing automatic code completion.

Once the object variables have been declared, you must specifically assign the application variable to the Crystal Reports Automation Server and choose a particular .RPT report file for the report variable. This is accomplished by using the Set statement. For the Application object, use the VB CreateObject method, passing the class argument "Crystal.CRPE.Application". You can then use the Application object's OpenReport method to assign a specific .RPT file to the Report object. The Set statements should appear in Form_Load, Sub Main, or another appropriate location, depending on the scope you want the object variables to have throughout the project. In the Xtreme Orders sample application, these objects are assigned in Form_Load:

```
'Assign application object
Set Application = CreateObject("Crystal.CRPE.Application")
'Open the report
Set Report = Application.OpenReport(App.Path & "\Xtreme Orders.rpt")
```

At the end of your application (in Form_Unload, for example), it is recommended that you set the object variables to Nothing to release memory:

```
Set Report = Nothing
Set Application = Nothing
```

Controlling the Preview Window

After you assign the Report object to an .RPT file, then the “fun” can begin. In its simplest form, you simply display the report in a preview window with one call. Execute the Report object’s Preview method to show the report in a preview window:

```
Report.Preview("Xtreme Orders Report")
```

The Preview method accepts several arguments, including the “Xtreme Orders Report” string that appears in the preview window’s title bar.

If you compare the Automation Server with the RDC, discussed in Chapter 23, you’ll notice you aren’t dealing with a separate Smart Viewer ActiveX object—the Automation Server includes its own built-in preview window. However, you’ll still probably want to be able to customize the behavior of the preview window. In addition to extra arguments that can be provided to the Preview method to control the window title, size, and position, a whole other set of properties is available that controls which elements of the preview window appear or can be interacted with.

The PrintWindowOptions object resides below the Report object and contains a set of properties that controls the preview window’s appearance and behavior. You can control which buttons appear and whether or not drilling down is enabled. In the Xtreme Orders sample application, two of these properties are set to control drilling down and the display of the group tree:

```
Report.PrintWindowOptions.HasGroupTree = True
Report.PrintWindowOptions.CanDrillDown = True
```

There are other properties of the PrintWindowOptions object that you can set, as well. Look at the Object Browser or Developer’s Help for more information.

Controlling Record Selection

One of the obvious reasons to integrate Crystal Reports with Visual Basic in the first place is to completely control the user interface. You'll no doubt want to control report record selection, based on a user's selections made using your own user interface. Because VB has such powerful user-interface features (such as the VB date picker and other custom controls), you will have much more flexibility using VB controls than you will with Crystal Reports parameter fields.

Because the record-selection formula you pass to the Automation Server must still conform to Crystal Reports syntax, you need to know how to create a Crystal Reports formula. If you are used to using the Crystal Reports Select Expert for record selection, you need to familiarize yourself with the actual Crystal Reports formula that it creates, before you create a selection formula in your VB application. A Crystal Reports record-selection formula is a Boolean formula that narrows down the database records that will be included in the report. For example, the following record-selection formula will limit a report to orders placed in the first quarter of 1997 from customers in Texas:

```
{Orders.Order Date} In Date(1997,1,1) To Date(1997,3,31)
And {Customer.Region} = "TX"
```

Tip: For complete discussions of Crystal Reports record selection and how to create Boolean formulas, consult Chapters 5 and 6, respectively.

Use the RecordSelectionFormula property of the Report object to set the record-selection formula for the report. You can set the property to either a string expression or a variable. Here's the code from the sample application that sets record selection, based on the contents of the user-supplied From Date and To Date text boxes:

```
'Supply record selection based on dates
strSelectionFormula = "{Orders.Order Date} in Date(" & _
Format(txtFromDate, "yyyy,m,d") & ") to Date(" & _
Format(txtToDate, "yyyy,m,d") & ")"
Report.RecordSelectionFormula = strSelectionFormula
```

Record-Selection Formula Tips

You need to keep in mind several important points when building a Crystal Reports record-selection formula within your application. Specifically, you can employ some tricks to make sure that your string values are formatted properly, and that as much of the SQL selection work as possible is done on the server rather than on the PC.

The string value you pass must adhere *exactly* to the Crystal Reports formula syntax. This includes using correct Crystal Reports reserved words and punctuation. The example from the sample application shows the necessity of building a Date(yyyy,mm,dd) string to pass to the selection formula.

Also, it's easy to forget the required quotation marks or apostrophes around literals that are used in comparisons. For example, you may want to pass the following selection formula:

```
{Customer.Region} = 'TX' And {Orders.Ship Via} = 'UPS'
```

If the characters TX and UPS are coming from controls, such as a text box or combo box, consider using the following VB code to place the selection formula in a string variable:

```
strSelectionFormula = "{Customer.Region} = " & txtRegion & _
" And {Orders.Ship Via} = " & cboShipper
```

At first glance, this appears to create a correctly formatted Crystal Reports record-selection formula. However, if you pass this string to the RecordSelectionFormula property, the report will fail when it runs. Why? The best way to troubleshoot this issue is to look at the contents of strSelectionFormula in the VB Immediate window, by setting a breakpoint, or by using other VB debugging features. Doing so will show that the contents of the string variable after the preceding code executes as follows:

```
{Customer.Region} = TX And {Orders.Ship Via} = UPS
```

Notice that no quotation marks or apostrophes appear around the literals that are being compared in the record-selection formula—a basic requirement of the Crystal Reports formula language. The following VB code will create a syntactically correct selection formula:

```
strSelectionFormula = "{Customer.Region} = '" & txtRegion & _
"' And {Orders.Ship Via} = '" & cboShipper & "'"
```

If your report will be using a SQL database, remember also that the Automation Server will attempt to convert as much of your record-selection formula as possible to SQL when it runs the report. The same caveats apply to the record-selection formula you pass from your VB application as apply to a record-selection formula you create directly in Crystal Reports. In particular, using built-in Crystal Reports formula functions, such as UpperCase or IsNull, and using OR operators instead of AND operators, will typically cause record selection to be moved to the client (the PC) instead of the database server. The result is very slow report performance. To avoid this situation, take the same care in creating record-selection formulas that you pass from your application as you would in Crystal Reports. Look for detailed discussions on record-selection performance in both Chapters 6 and 14.

You can also choose to create the SQL statement that the report will use right in your VB application. You can then submit it to the report by setting the Report object's SQLQueryString property.

Setting Formulas

Another major benefit of report integration with Visual Basic is that you can change Crystal Reports formulas at run time. This is useful for changing formulas related to groups that might also be changed from within your code, formulas that show text on the report, or formulas that relate to math calculations or conditional formatting on the report.

Setting formulas at run time is similar to setting the record-selection formula at run time (as described in the previous section). You need a good understanding of the Crystal Reports formula language to adequately modify formulas inside your VB code. If you need a refresher on Crystal Reports formulas, refer to Chapter 5.

The Automation Server exposes a FormulaFields collection in the Report object that can be read from or written to at run time. This collection returns multiple FormulaFieldDefinition objects, one for each formula defined in the report. Although a great deal of information can be gleaned from the FormulaFieldDefinition object, the simplest way to change the contents of a formula is simply to set the Text property for the particular member of the collection that you want to change. The collection is one-based, with each member corresponding to its position in the list of formula fields. Unlike the RDC, the Automation Server lets you pass a string value as the index to the FormulaFields collection. This is more convenient, letting you call the formula field you want to change by name, instead of by number.

In the Xtreme Orders sample application, the Order + Tax formula is modified based on the user's input in the Tax Rate text box. The formula is changed using the following code:

```
'Set @Order + Tax formula
If txtTaxRate = "" Then
   Report.FormulaFields("Order + Tax").Text = _
   "{Orders.Order Amount}"
Else
   Report.FormulaFields("Order + Tax").Text = _
   "{Orders.Order Amount} *" & Str(txtTaxRate / 100 + 1)
End If  'txtTaxRate = ""
```

Notice that the particular member of the collection that you want to change can be specified by the string "Order + Tax". Make sure you don't include the @ sign when specifying the formula name. Since Order + Tax is the first formula in the formula list, you can also change the formulas by using an index number, as in the following code:

```
Report.FormulaFields(1).Text = "{Orders.Order Amount}"
```

The Text property requires a string expression or variable, correctly conforming to the Crystal Reports formula syntax, that contains the new text you want the formula to contain. Take care when assigning a value to the formula. The Order + Tax formula is defined in Crystal Reports as a numeric formula, so you should pass it a formula that will evaluate to a number.

If the user leaves the text box on the form empty, the VB code assumes there is no additional sales tax and simply places the Order Amount database field in the formula—the formula will show the same amount as the database field. If, however, the user has specified a tax rate, the VB code manipulates the value by dividing it by 100 and then adding 1. This creates the proper type of number to multiply against the Order Amount field, to correctly calculate tax. For example, if the user specifies a tax rate of 5 (for 5 percent tax), the VB code will change the value to 1.05 before combining it with the multiply operator and the Order Amount field in the formula.

If you're contrasting the Automation Server with the RDC, you'll also become aware that you can't change the contents of Text objects with the Automation Server at run time. So, changing textual information, such as an information message that contains selection options the user made, must be done by modifying a string formula at run time.

In the Xtreme Orders sample application, the @Sub Heading string formula appears in the page header and contains a description of the report options that are chosen at run time. This formula is set in the sample application based on selections the user has made on the form. Here's the sample code:

```
'Set @Sub Heading formula
strSubHeading = "'" & txtFromDate & " through " & txtToDate
strSubHeading = strSubHeading & ", By " & cboGroupBy
If txtTaxRate = "" Then
   strSubHeading = strSubHeading & ", No Tax'"
Else
   strSubHeading = strSubHeading & ", Sales Tax = " & _
   txtTaxRate & "%'"
End If  'txtTaxRate = ""
Report.FormulaFields("Sub Heading").Text = strSubHeading
```

Here, the string variable strSubHeading has been declared earlier in the procedure and is used as a work string to build the formula contents. It's actually possible to just use the Text property from the FormulaFieldDefinition object in the same fashion, because it's both a read and write property.

Note: You needn't change the property for all formulas in the report—just the ones you want to modify at run time.

Passing Parameter Field Values

The .RPT file that the Report object is bound to may contain parameter fields that are designed to prompt the user for information whenever the report is refreshed. If you simply leave the parameter fields as is, the application automatically prompts the user for those values when the report runs. However, you'll probably want to supply these parameter values through code, based on your own user interface.

Since you can control both report record selection and report formulas at run time from within your VB application, there isn't as much necessity to use parameter fields with an integrated report as there might be for a stand-alone report running with regular Crystal Reports. However, you may be "sharing" a report on a network or via another sharing mechanism. In this situation, you might want to maintain parameter fields in the .RPT file to accommodate report viewers who won't be using your front-end application to run the report. Or, using existing parameter fields may just be simpler than designing extra code to accomplish the same thing.

In these situations, use the Report object's ParameterFieldDefinitions collection to manipulate the parameters. As with formulas, there is one ParameterFieldDefinition object in the collection for each parameter field defined in the report. And, as with formulas, the proper member of the collection is retrieved by using as an index either the actual parameter field name or a one-based index determined by the order in which the parameter fields were added to the report. Unlike the RDC, you can retrieve a parameter field by using its name as the index (without the question mark). To set the value of a parameter field, use the ParameterFieldDefinition object's SetCurrentValue method. This passes a single value to the report parameter field.

In the Xtreme Orders sample application, the ?Highlight parameter field is used to conditionally format the details section when the order amount exceeds a parameter supplied by the viewer. Here is the sample code to correctly pass the value to the parameter field:

```
'Set ?Highlight parameter value
If txtHighlight = "" Then
   Report.ParameterFields("Highlight").SetCurrentValue (0)
Else
   Report.ParameterFields("Highlight").SetCurrentValue _
   (Val(txtHighlight))
End If  'txtHighlight = ""
```

Here, the value is set to 0 if the text box is empty. This is important to note, because the actual parameter field in the report is numeric. Passing it an empty string might cause a run-time error when the report runs. Also, because the SetCurrentValue method is executed at least once, and a value has been passed to the parameter field, the user will no longer be prompted for it when the report is run.

Note: The Crystal Reports 8 Automation Server Object Model has not been updated for Version 8 and does not expose many newer features of parameter fields, such as range values, multiple values, and edit masks. If the .RPT file you are integrating depends on these features, you may need to redesign the report to accommodate the Automation Server, or consider using the RDC to integrate the report.

Manipulating Report Groups

You'll often want to add flexibility to your application by allowing users to choose how they want their reports grouped. By enabling your end users to change these options from within an application at run time, you can provide them great reporting flexibility. In many cases, you can create the appearance that several different reports may be chosen based on user input. In fact, the application will be using the same Report object, but grouping will be changed at run time, based on the user's input.

For example, part of the Xtreme Orders sample application is a combo box on the Print Report form that lets the user choose between Quarter and Customer grouping. Based on this setting, you'll want your application to choose the field on which the first (and only) report group is based, as well as choose quarterly grouping if the Order Date field is chosen. To familiarize yourself with various grouping options, refer to Chapter 3.

Manipulating groups in the Automation Server requires you to navigate "down the object hierarchy." Ultimately, the database or formula field a group is based on is specified by the GroupConditionField property of the Area object. A report contains an Areas collection containing an Area object for each area in a report. Consider an "area" of the report to be each individual main section of the report, such as the page header, details section, group footer, and so on. If there are multiple sections on the report, such as Details a and Details b sections, they are each sections in the overall details *area,* and only one member of the Areas collection exists for all of those sections. The index that is used to refer to an individual member of the collection can be a string value or a numeric value. This allows you to specify "RH", "PH", "GH*n*", "D", "GF*n*", "PF", or "RF" as index values or a one-based index number. Using the string values makes your code much easier to understand when working with the Areas collection.

Once you navigate to the specific Area object to specify the GroupConditionField property, you are faced with a peculiarity of the Automation Server Object Model. With other integration methods (the ActiveX control, in particular), you supply the actual field name (such as {Orders.Order Date} or {Customer.Customer Name}) to indicate which field you wish to base a group on. The Automation Server doesn't make life this simple. You must specify a DatabaseFieldDefinition object to the GroupConditionField property—you can't just pass a field name.

DatabaseFieldDefinition is found "deep" in the object hierarchy—there is one of these objects for each field contained in the DatabaseFieldDefinitions collection in a DatabaseTable object. And, multiple DatabaseTable objects are contained in the DatabaseTables collection of the Database object, contained inside the overall Report object. However, unlike the RDC, which shares a similar object hierarchy, the collections just mentioned will accept a string index value or a one-based number.

Although this sounds confusing and hard to navigate, you'll eventually be able to travel through the object hierarchy fairly quickly to find the database table and field you want to provide to the GroupConditionField property. Look at the following code from the Xtreme Orders sample application:

```
'Set grouping
If cboGroupBy = "Quarter" Then
   'The quarterly grouping will only work if the report is set
   'to "Convert Date/Time Fields to Date" - known bug
   'with SCR 7 and 8
   Report.Areas("GH").GroupOptions.ConditionField = _
      Report.Database.Tables("Orders").Fields("Order Date")
   Report.Areas("GH").GroupOptions.Condition = crGCQuarterly
Else
   Report.Areas("GH").GroupOptions.ConditionField = _
   Report.Database.Tables("Customer").Fields("Customer Name")
   Report.Areas("GH").GroupOptions.Condition = crGCAnyValue
End If  'cboGroupBy = "Quarter"
```

Caution: Notice the comments in the code relating to quarterly grouping. A bug in Crystal Reports 7 and 8 returns an Automation error if you try to group on a date/time field. However, if you open the report in the Crystal Reports designer and choose File | Report Options and choose To Date as the Convert Date/Time Field option, quarterly grouping will work properly. The trade-off is that true date/time data types won't be available for use in the report—they will be converted to date-only fields.

In this example, notice that the grouping is based on the user choice for the cboGroupBy combo box. In either case, the GroupConditionField property of the "GH" member of the Areas collection (the group header) is being set. If the user chooses to group by Order Date, the property is set to the "Order Date" member of the Fields collection in the "Orders" member of the Tables collection. If the user chooses to group by Customer Name, the Customer Name field in the Customer table will be used for grouping. Instead of specifying string indexes for these collections, you can use one-based numbers, if you choose.

In addition to choosing the database field you want to use for the group, you may have to choose how the grouping occurs, particularly if you use a date or Boolean field as the grouping field. This is accomplished by manipulating the GroupAreaOptions object below the Area object. This object contains a Condition property that can be set to an integer value or to an Automation Server–supplied constant (see the explanation for the GroupAreaOptions object in online Help for specific constants and integer values). In the sample code shown previously, the group is set to break for every calendar quarter if Order Date grouping is set (hence, the crGCQuarterly constant that is provided). If Customer Name grouping is chosen, the crGCAnyValue constant is supplied, indicating that a new group will be created whenever the value of the group field changes.

Caution: Automatic code completion won't work when you navigate deep in the object hierarchy, as demonstrated in the code shown earlier in the chapter. The Automation Server helps you complete code only one level deep, so keep Developer's Help or the Object Browser handy to help you navigate through objects this deep in the hierarchy. An alternative mentioned in the online Developer's Help is to use Dim statements to create additional object variables for each of the objects that is below the first level in the hierarchy, and then use multiple Set statements to assign them values. While this will let you use automatic code completion, it also creates monstrous amounts of extra code.

Conditional Formatting and Formatting Sections

If you'll be presented reports in the Automation Server preview window, you'll want to take advantage of as many of Crystal Reports' interactive reporting capabilities as possible. One particular benefit of online reporting is the drill-down capability. This allows a report to be presented initially at a very high summary level, showing just subtotals or grand totals, or perhaps just a pie chart or bar chart that graphs high-level numbers. The viewer can then double-click a number or chart element that interests them, to "drill down" to more-detailed numbers, eventually reaching the report's details section. These levels of drilling down are limited only by the number of report groups you create on your report.

The Xtreme Orders sample application gives the user the opportunity to specify whether or not to see the report as a summary report only. When the Summary Report Only check box is selected, the VB application needs to hide the report's details section so that only group subtotals appear on the report. In addition, you need to control the appearance of the report's two page header sections (Page Header a and Page Header b), as well as the two Group Header #1 sections (Group Header #1a and Group Header #1b).

This page header and group header formatting is to accommodate two different sections of field titles that you wish to appear differently if the report is presented as a summary report instead of a detail report. If the report is being displayed as a detail report, the field titles should appear at the top of every page of the report, along with the subheading and smaller report title. If the report is displayed as a summary report, you only want the field titles to appear in the group header section of a drill-down tab when the user double-clicks a group. Since no detail information will be visible in the main report window, field titles there won't be meaningful.

And, finally, you'll want to show Group Header #1a, which contains the group name field, if the report is showing detail data. This indicates what group the following set of orders applies to. However, if the report shows only summary data, then showing both the group header and group footer is repetitive—the group footer already contains the group name, so showing the group header, as well, looks odd. But, you will want the group header with the group name to appear in a drill-down tab.

Therefore, you need to control the appearance of four sections when a user chooses the Summary Report Only option. Table 1 gives a better idea of how you'll want to conditionally set these sections at run time.

Section	**Detail Report**	**Summary Report**
Page Header b (field titles)	Shown	Suppressed (no drill-down)
Group Header #1a (group name field)	Shown	Hidden (drill-down okay)
Group Header #1b (field titles)	Suppressed (no drill-down)	Hidden (drill-down okay)
Details (detail order data)	Shown	Hidden (drill-down okay)

Table 1: Section Formatting for Different Reports

Notice that, in some cases, sections need to be completely suppressed (so they don't show up, even in a drill-down tab) as opposed to hidden (which means they will show up in a drill-down tab, but not in the main Preview tab). There is a somewhat obscure relationship among areas and sections that makes this determination. Specifically, only complete areas (such as the entire Group Header #1) can be hidden, not individual sections (for example, Group Header #1a). However, both areas and sections can be suppressed individually.

This presents a bit of a coding challenge for the Xtreme Orders sample application. First, you need to determine the correct combination of area hiding and section suppressing that sets what appears in the preview window and in a drill-down tab. If an area is hidden, none of the sections in it will appear in the main preview window. But, when the user drills down into that section, you will want to control which sections inside the drilled-down area appear in the drill-down tab. Once you've determined which areas you want to hide, you must find the correct property to set for that particular area. You must also find the correct property to suppress sections inside the hidden area, or to show other sections that aren't hidden.

This functionality is controlled by manipulating two different, but similar, sets of objects below the Report object. First is the Areas collection, which contains an Area object for each area in the report. For example, the page header is considered an individual area. An *area* encompasses one or more related sections—both Group Header #1a and Group Header #1b sections are combined in one single Group Header 1 area.

The Report object also contains a Sections collection, which contains a Section object for each section in the report. A *section* encompasses each individual section of the report, regardless of how many sections are inside a single area. For example, the page header is considered an individual section. But, even though Group Header #1a and Group Header #1b are considered a single area, they are separate sections.

Manipulating areas and sections can "send you deep" into the object hierarchy. However, choosing the right member of the collections is made easier by the option of providing a string value as the index. Online Help can help you determine which section and area abbreviations to use as indexes, or you can use the short section names that appear in the Crystal Reports designer when you have the Short Section Names option turned on in File | Options. For example, either the details area or details section member can be accessed by supplying an index of "D". The Group Header 1 area can be accessed by supplying an index of "GH1" to the Areas collection. Getting to the individual sections is possible by providing either "GH1a" or "GH1b" indexes to the Sections collection.

Once you've navigated to the correct section or area, you must set the options by manipulating the AreaOptions or SectionOptions object below each respective Area or Section object. These objects contain the actual properties you need to set, such as NotHideForDrillDown and Visible. Note that these two particular properties are "backward" from what you might be used to in other integration methods or in the Crystal Reports designer. If you want an area hidden for drill-down, you can't set a property to true. You must actually set the

NotHideForDrillDown property to false. Also, if you want an individual section suppressed, you have to set the SectionOptions object's Visible property to false.

Keeping in mind the previous discussion and the hide/suppress requirements outlined in Table 1, consider the following code from the sample application:

```
'Hide/show areas and sections
With Report
   If chkSummary Then
      .Areas("D").Options.NotHideForDrillDown = False
      .Sections("PHb").Options.Visible = False
      .Areas("GH1").Options.NotHideForDrillDown = False
      .Sections("GH1a").Options.Visible = True
      .Sections("GH1b").Options.Visible = True
      'NOTE! multiple-section letters in sections collection index
      '(e.g., GH1a) must be lowercase!
   Else
      .Areas("D").Options.NotHideForDrillDown = True
      .Sections("PHb").Options.Visible = True
      .Areas("GH1").Options.NotHideForDrillDown = True
      .Sections("GH1a").Options.Visible = True
      .Sections("GH1b").Options.Visible = False
      'NOTE! multiple-section letters in sections collection index
      '(e.g., GH1a) must be lowercase!
    End If  'chkSummary
End With  'Report
```

Caution: Notice the comments in the sample code that warn about case sensitivity of indexes provided to the Sections collection. "GH1a" will be correctly interpreted, while "GH1A" will result in a run-time error.

Notice that both Area objects for the details and Group Header #1 areas are being formatted, as well as individual sections. While you can navigate ever deeper into the Areas collection to eventually get to the sections inside, it's easier and less code-intensive to manipulate the Sections collection directly.

Choosing Output Destinations

By executing the Report object's Preview method, you can display the report contents in the Automation Server's built-in preview window. However, you'll certainly encounter other situations in which you want the report sent to a different destination. The Automation Server supports all the same output destinations as the Crystal Report designer's File | Print | Export command, including the printer, various file formats, and e-mail.

In the Xtreme Orders sample application, radio buttons and a text box allow the user to choose to view the report in the preview window, print the report to a printer, or attach the report as a Word document to an e-mail message. If the user chooses e-mail as the output destination, an e-mail address can be entered into a text box. Once that is done, you need to set the output destination automatically in your VB code, based on the user's selection.

If the user chooses to view the report on the screen, you'll use the Report object's Preview method. If you wish to print the Report object to a printer, execute the Report object's PrintOut method. The syntax is as follows:

```
object.PrintOut Prompt, Copies, Collated, StartPage, StopPage
```

In this syntax statement, *Prompt* is a Boolean value specifying whether or not to prompt the user for report options, *Copies* is a numeric value specifying how many copies of the report to print, *Collated* is a Boolean value specifying whether or not to collate the pages, and *StartPage* and *StopPage* are numeric values identifying which page numbers to start and stop printing on. The parameters are optional. If you want to skip any that come before a parameter you want to specify, just add commas in the correct positions for the skipped parameter, and don't include a value.

If you wish to export the report to a disk file or attach the report to an e-mail message as a file, use the Report object's Export method:

```
object.Export PromptUser
```

PromptUser is a Boolean value indicating whether or not you want to prompt the user for export options. If you choose not to prompt the user for export options, you need to set the properties of the report's ExportOptions object. The ExportOptions object contains many properties that determine the output destination (file or e-mail), the file format (Word, Excel, and so on), the filename, the e-mail address, and others. Many of these properties accept integer values or Automation Server–supplied descriptive constants. If you choose MAPI as the destination, the user’s PC needs to have a MAPI-compliant e-mail client installed, such as Microsoft Outlook or Eudora Pro. Look for all the available ExportOptions properties by searching the online Help for “ExportOptions Object (Crystal Report Engine Object Library).”

In the sample Xtreme Orders application, the following code is used to choose an output destination, as well as to specify a file type and e-mail information, based on user selections. (Note that this sample code directly utilizes the report’s Preview method, while the actual sample application on the CD accompanying this book sets a View object to the contents of the report’s Preview method. This is discussed later in the chapter, under “Handling Preview Window and Report Events.”)

```
'Set output destination
If optPreview Then Report.Preview("Xtreme Orders Report")
If OptPrint Then Report.PrintOut (True)
If OptEmail Then
   With Report
       .ExportOptions.DestinationType = crEDTEMailMAPI
       .ExportOptions.MailToList = txtAddress
       .ExportOptions.MailSubject = "Here's the Xtreme Orders Report"
       .ExportOptions.MailMessage = _
"Attached is a Word document with the latest Xtreme Orders Report."
       .ExportOptions.FormatType = crEFTWordForWindows
       .Export (False)
   End With  'Report
End If  'optEmail
```

Caution: If you use the Report object’s Export method to export or send the report attached to an e-mail message, make sure you specify all the necessary ExportOptions object properties. If you fail to specify necessary properties, or specify them incorrectly, an error may be thrown. Or, the user may be prompted to specify export options, even though you specifically added a “false” argument to the Export method. If the export takes place at an unattended machine, this will cause your application to wait for a response when no user is available to respond.

Changing the Data Source at Run Time

Obviously, one of the main benefits of integrating a Crystal Report in a Visual Basic program is the ability to tie the report's contents to a data set or result set that the program is manipulating. This allows a user to interact with your application and then run a report that contains the same set of data that the user has selected.

One way to accomplish this is to create a Crystal Reports record-selection formula or a SQL statement based on your user interface. The formula or SQL statement can then be passed to the report (as discussed later in this chapter). This will allow the user to interact with a data grid, form, or other visual interface in the application, and then create a report based on its contents.

However, you may encounter situations in which you prefer to have the contents of an actual VB record set from one of Visual Basic's intrinsic data models act as the data source for a report. The Automation Server SetPrivateData method allows you to change at run time the data source that a table is based on. Using SetPrivateData, you can initially design the report using a particular "Active Data" database or data source. Then, you can pass a Visual Basic DAO, RDO, or ADO record set, snapshot, or result set to the report at run time. The Automation Server will use the current contents of the VB data set to populate the report.

The syntax for SetPrivateData is as follows:

```
object.SetPrivateData DataTag, Data
```

In this statement, *object* is a DatabaseTable object that you want to supply data to, *DataTag* indicates the type of data you are passing to the object (the only allowable value for *DataTag* is 3), and *Data* is the DAO, ADO, or RDO data set that you want to have supply data to the report.

SetPrivateData applies to the Automation Server's DatabaseTable object, several levels deep in the object hierarchy. This object is below the Database object, which falls below the Report object. The Database object contains a collection of DatabaseTable objects—one for each table used in the database. For example, suppose you have declared a Report object that contains a database with several tables, including a table named Customer. You also have defined an ADO Resultset object elsewhere in your VB application with the name ADOrs. The following code will pass the contents of the result set to the report. When the report is previewed, printed, or exported, the contents of the result set will be used by the report in place of the Customer table.

```
Report.Database.Tables("Customer").SetPrivateData 3, ADOrs
```

One important issue to remember is that of field names. Make sure the data source you pass to the report using SetPrivateData contains the same field names and data types as the original data source. Otherwise, the report may fail or return unexpected results.

Caution: If you use SetPrivateData to change the data source for a database table object, make sure the data source you pass to the report remains in scope whenever the report runs. For example, you may define a Report object that's global to your application (by declaring it at the module level), but if you pass it a data source that was declared at the form level and then try to run the report in another form, the data source will be out of scope and the report will fail. The ReadRecords method is available to read the contents of the data source into the Report object, if maintaining scope for the data source isn't practical. Search Developer's Help for information on ReadRecords.

Handling Preview Window and Report Events

One powerful feature that the Automation Server shares with the RDC (discussed in Chapter 23) is the ability to trap events that occur in the preview window, as well as trapping certain events that the Report object creates. This allows the developer to trap button clicks, report-processing cycles, and drill-down attempts. These events can be used to modify the behavior of the preview window or report, as well as to execute some additional supplementary code when a certain event occurs.

Report Object Event Handling

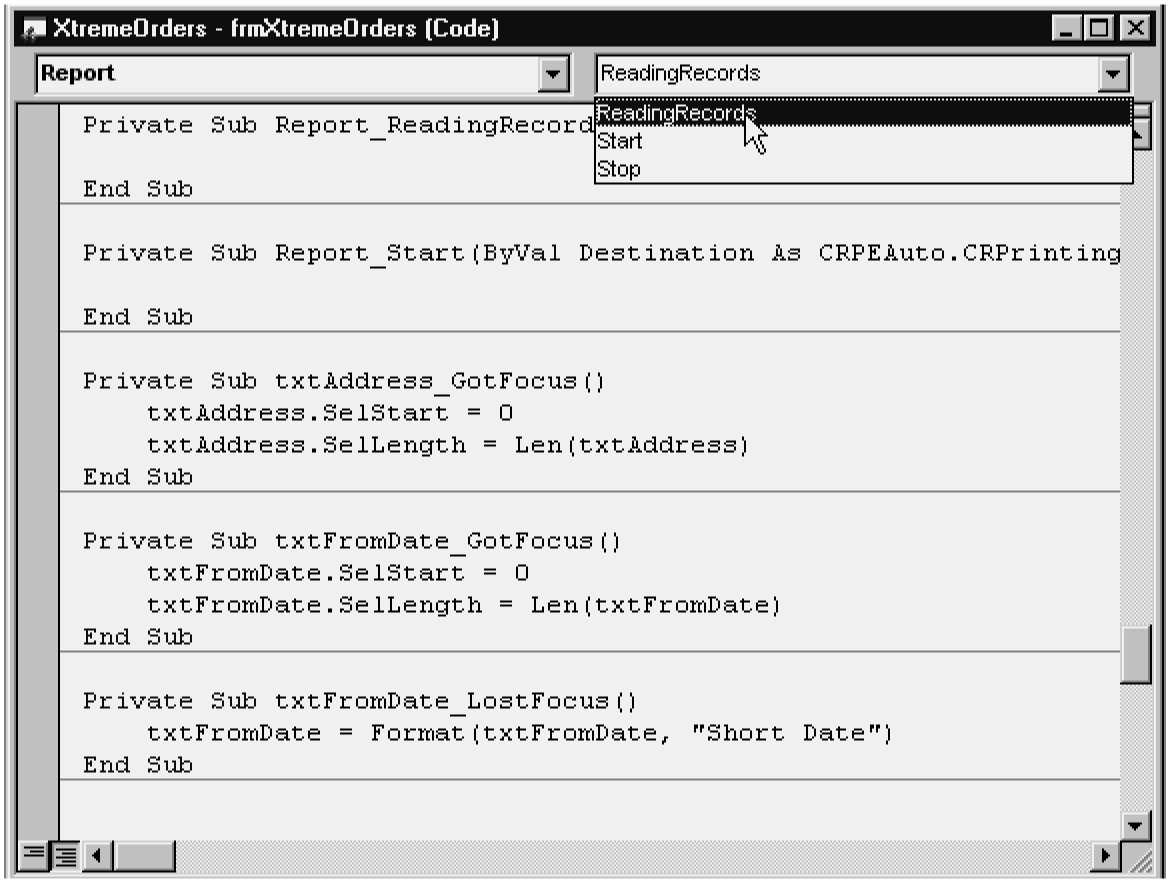
To enable Report object events, the Report object must be declared Public and WithEvents. So, instead of using the following declaration:

```
Dim Report As CRPEAuto.Report
```

you would use:

```
Public WithEvents Report As CRPEAuto.Report
```

The WithEvents parameter will add the Report object name to the object drop-down list in the upper left of the Code window. When you choose the Report object, the following four events can be chosen in the procedure drop-down list in the upper right of the Code window:



```
Private Sub Report_ReadingRecord

End Sub

Private Sub Report_Start(ByVal Destination As CRPEAuto.CRPrinting

End Sub

Private Sub txtAddress_GotFocus()
    txtAddress.SelStart = 0
    txtAddress.SelLength = Len(txtAddress)
End Sub

Private Sub txtFromDate_GotFocus()
    txtFromDate.SelStart = 0
    txtFromDate.SelLength = Len(txtFromDate)
End Sub

Private Sub txtFromDate_LostFocus()
    txtFromDate = Format(txtFromDate, "Short Date")
End Sub
```

- **Start** Occurs when the report starts processing
- **Stop** Occurs when the report processing is finished
- **FieldMapping** Occurs while verifying database, if the database has changed

- **ReadingRecords** Occurs when the report begins reading database records

You can add code in these individual event modules to notify the user that processing has started or stopped, the database has changed, or to intervene when the report begins reading records.

Preview Window Event Handling

Although report event handling can be helpful, you'll probably get more use out of preview-window event handling. This allows your code to intercept button clicks, drill-downs, and other preview-window interactions. You can then modify aspects of the report or execute supplementary code based on the action.

Preview-window event handling is accomplished by declaring and using two new object types not previously discussed: the View object and the Window object. The *View* object is equivalent to a tab in the preview window. When the report is first previewed, the main Preview tab is considered a View object. If a viewer double-clicks a group and creates a new drill-down tab, that new tab is also considered to be a View object. While multiple View objects can exist at any one time, only one can actually be manipulated—the currently selected tab.

View objects are always contained inside the *Window* object. Generally, the entire preview window can be considered to be the single Window object, with the multiple View objects (the main Preview tab and drill-down tabs) all appearing inside it. The Window object cannot actually be declared directly. It must be declared using the Parent property of a View object after the View object has been assigned to a particular Report object.

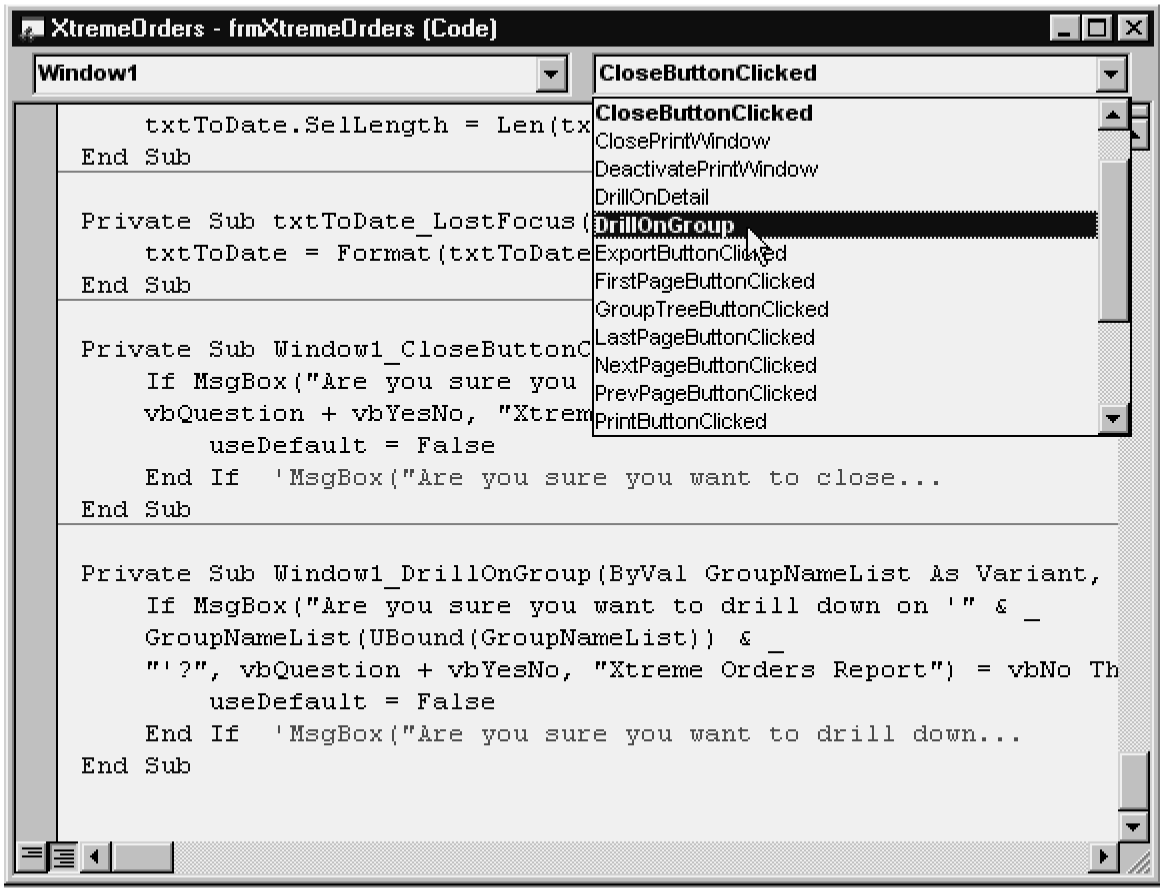
Just as you have to declare a Report object to be Public and WithEvents to enable event trapping, the Window object must also be declared using these two options. Because the actual View object doesn't "fire" any events, it doesn't have this requirement. Once the object variables have been declared, the View object must be assigned to the Preview method of a Report object. Then, the Window object is assigned as the parent of the View object. Look at the following sample code:

```
Dim View1 As CRPEAuto.View
Public WithEvents Window1 As CRPEAuto.Window
```

```
Set View1 = Report.Preview("Xtreme Orders Report")
Set Window1 = View1.Parent
```

Here, the object variable View1 is declared to be of the View class, and Window1 is declared to be of the Window class (both are fully qualified to refer to the CRPEAuto Automation Server). Notice that Window1 is declared Public and WithEvents. Then, the View1 object variable is pointed to the main Preview tab of the report by assigning it to the Preview method of the Report object. Finally, the Window1 object variable is set to the whole preview window by assigning it to the Parent property of the View object.

As with the Report object, once the Window1 object has been declared "WithEvents," an additional object appears in the object drop-down list in the upper left of the Code window. When you choose the Window1 object, you'll notice several events available in the procedure drop-down list in the upper right of the Code window.



```
    txtToDate.SelLength = Len(tx
End Sub

Private Sub txtToDate_LostFocus(
    txtToDate = Format(txtToDate
End Sub

Private Sub Window1_CloseButtonC
    If MsgBox("Are you sure you
    vbQuestion + vbYesNo, "Xtrem
        useDefault = False
    End If  'MsgBox("Are you sure you want to close...
End Sub

Private Sub Window1_DrillOnGroup(ByVal GroupNameList As Variant,
    If MsgBox("Are you sure you want to drill down on '" & _
    GroupNameList(UBound(GroupNameList)) & _
    "'?", vbQuestion + vbYesNo, "Xtreme Orders Report") = vbNo Th
        useDefault = False
    End If  'MsgBox("Are you sure you want to drill down...
End Sub
```

As you can see, there is a large number of preview-window events that can be trapped. The Xtreme Orders sample application has only two simple examples of these capabilities. When the user drills down on a group, a confirmation message box appears asking the user to confirm the drill-down. If the answer is Yes, the drill-down will occur. If the answer is No, the drill-down will be canceled. And, after a user has drilled down and then clicks the close view button (the red x) in the viewer, another confirmation message dialog box appears asking them to confirm the closure. As with the original drill-down, the drill-down tab will be closed only if the user answers Yes. Otherwise, the event will be canceled.

The initial drill-down is trapped by the Window object's DrillOnGroup event. Every time a group name, summary, or subtotal is double-clicked, this event will "fire" before the actual drill-down occurs. Several parameters are passed into the event:

- **GroupNameList** Contains an array of the group names that are being drilled into
- **DrillType** Indicates what type of drill-down is occurring (whether the view drilled down on a group or graph, or used the group tree)
- **useDefault** A Boolean value that can be set to determine whether the drill-down actually occurs
- **ReportName** Documented as "reserved for future use"—it still serves no purpose in Version 8

One important part of the drill-down example is actually determining which group will be affected when the drill-down occurs. This ability may be important to fully exploit the power of the event model—trapping a drill-down may be of limited use if you can't determine which group is actually being manipulated. The Window object's DrillOnGroup event makes this task easier by passing the GroupNameList parameter into the event function. This zero-based array contains an element for each group prior to the group that is being drilled into, as well as the current group just drilled into. If your report contains only one level of grouping (as does the Xtreme Orders sample report), the GroupNameList will always contain just element 0, that of the group that was drilled into.

However, if your report contains multiple levels of grouping, you'll be able to determine how many levels of drill-down have occurred and which groups have been drilled down on to get to this point. For example, if your report is grouped on three levels, Country, State, and then City, a user can first drill down on Canada. This will fire the DrillOnGroup event, and the GroupNameList array will contain only element 0, "Canada." Then, in the Canada drill-down tab, the user might double-click BC. This will again fire the DrillOnGroup event, and the GroupNameList array will now contain two elements, "Canada" in element 0 and "BC" in element 1. Then, if in the BC drill-down tab the user double-clicks Vancouver, yet another DrillOnGroup event will fire. Inside this event the GroupNameList array will contain elements 0 through 2, "Canada," "BC," and "Vancouver."

If you are concerned about the lowest level of grouping only, just query the last element of the GroupNameList array using the VB UBound function. Even though the Xtreme Orders report contains only one level of grouping (thereby always allowing GroupNameList(0) to be used to determine the group), the UBound function has been used for upward-compatibility.

The other argument of note passed to the event is useDefault. This Boolean value can be set to true or false inside the event to determine whether the drill-down actually occurs. If useDefault is set to true (or left as is), the drill-down will occur. If it is set to false inside the DrillOnGroup event, the drill-down will be canceled and no drill-down tab will appear for the group in the viewer.

Here's the sample code from the Xtreme Orders application. Note that the actual group being drilled down on is included in the message box and that the results of the message box determine whether the drill-down occurs or not.

```
Private Sub Window1_DrillOnGroup(ByVal GroupNameList As Variant, _
ByVal DrillType As CRPEAuto.CRDrillType, useDefault As Boolean, _
ByVal ReportName As Variant)
   If MsgBox("Are you sure you want to drill down on '" & _
   GroupNameList(UBound(GroupNameList)) & _
   "'?", vbQuestion + vbYesNo, "Xtreme Orders Report") = vbNo Then
      useDefault = False
   End If  'MsgBox("Are you sure you want to drill down...
End Sub
```

The other Window event that is used in the Xtreme Orders sample application is CloseButtonClicked. This event fires whenever a user clicks the red × in the preview window to close the current drill-down tab. Again, the sample application simply displays a confirming message box asking whether or not the user really wants to close the drill-down tab. By setting the UseDefault parameter inside the event code, the drill-down tab is or is not closed, based on the user's choice.

Unlike the DrillOnGroup event, the Automation Server does not provide a string value indicating the drill-down tab that is being closed. The parameter being passed is ViewIndex, which returns an integer indicating the drill-down tab that is being closed (1 indicates the first drill-down tab, 3 the third, and so on). Unlike the RDC, there is no way for the Automation Server to determine the actual group name of the drill-down tab that's being closed.

Examine the following code from the Xtreme Orders sample application:

```
Private Sub Window1_CloseButtonClicked(ByVal ViewIndex As Integer, _
useDefault As Boolean)
   If MsgBox("Are you sure you want to close the drill down tab?", _
   vbQuestion + vbYesNo, "Xtreme Orders Report") = vbNo Then
      useDefault = False
   End If  'MsgBox("Are you sure you want to close...
End Sub
```

This event fires every time the close button in the preview window is clicked. If the viewer responds No to the message box, useDefault is set to false, thereby canceling the close event and still leaving the drill-down tab visible.

Enabling or Disabling Events with EventInfo

The Report object contains an EventInfo object below it. The EventInfo object contains several properties that you can set to true or false to control which report and preview-window events will fire. Even if you've declared a Window object to be "WithEvents," you may wish to occasionally restrict events from firing at certain times. EventInfo properties can be set to false to prevent the events from being trapped.

For example, to prevent the DrillOnGroup event from occurring, set the EventInfo GroupEventEnabled property to false. And, to disable the CloseButtonClicked event, set the EventInfo PrintWindowButtonEventEnabled property to false. Although this is not used in the Xtreme Orders sample application, the code would appear as follows:

```
Report.EventInfo.GroupEventEnabled = False
Report.EventInfo.PrintWindowButtonEventEnabled = False
```

In these situations, the events won't fire, but the actual action still occurs. In other words, if both of these lines of code are added to the sample application, the drill-down will occur without ever firing the event, and a new drill-down tab will appear without any user confirmation. When the close button is clicked, the drill-down tab will be closed, but the user won't ever be presented with the options to cancel the close operation.

Tip: Despite online Help that may indicate otherwise, the "enabled" properties of EventInfo are all read/write. They can be read or set at run time, not just read.

Error Handling

As with any Visual Basic program, you'll want to prepare for the possibility that errors may occur. Integrating Crystal Reports with VB requires that you anticipate errors that the Automation Server might encounter, in addition to other errors that the rest of your program may produce.

Automation Server errors are trapped in an On Error Goto handler, just like other VB errors. In many situations, you'll see typical VB-type error codes for report-related errors. For example, supplying an incorrect index argument to a Report object collection won't actually generate an Automation Server–specific error. Subscript Out of Range will be returned instead. Other error codes returned by the Automation Server will be in the 20*XXX* range. By using members of the Errors collection, such as Err.Number and Err.Description, you can handle errors or present meaningful error messages to the user. The *XXX* three-digit codes are specific to the Crystal Reports Print Engine, which is actually called by the Automation Server. For a complete breakdown of these error codes, search Developer's Help for "error codes, Crystal Report Engine."

Examine the following code from the Xtreme Orders sample application. Notice that this only traps two particular reporting errors; when an invalid e-mail address is provided, or when the viewer cancels report printing or exporting. If another error occurs, this routine will simply display the error code and text in a message box.

```
cmdOK_Click_Error:
Select Case Err.Number
   Case 20541
      MsgBox "Invalid e-mail address or other e-mail/export problem", _
             vbCritical, "Xtreme Orders Report"
      txtAddress.SetFocus
   Case 20545
      MsgBox "Process canceled", vbInformation, "Xtreme Orders Report"
   Case Else
      MsgBox "Error " & Err.Number & " - " & Err.Description, _
             vbCritical, "Xtreme Orders Report"
End Select  'Case Error.number
```

Caution: Often, you'll introduce errors before processing the report with the Preview, Printout, or Export methods, such as submitting a syntactically incorrect formula or using an incorrect section name when formatting sections. However, these statements won't result in an error. Typically, no error will be detected until you actually process the report with one of the three processing methods.

Other Automation Server Properties and Methods

Although this chapter has covered most common Automation Server procedures for handling report-customization requirements, several other areas warrant some discussion, particularly Automation Server functions that handle SQL database reporting, and functions that deal with subreports. In addition, the Automation Server DiscardSaveData method needs to be addressed, which should be used to "reset" or clear the contents of properties that may have been set in previous VB code.

The DiscardSavedData Method

A straightforward feature of the Crystal ActiveX control discussed in Chapter 22 is the Reset method. This method comes in handy when you may run a report more than once inside the application without explicitly closing and redeclaring the report. The ActiveX Reset method returns all properties to the values they originally had in the .RPT file that is being printed. Although no similar method exists for the Automation Server, you will still want to use the Report object's DiscardSavedData method if you will be running a report multiple times without setting the Report object to "Nothing" and redeclaring it.

The Xtreme Orders sample application is a good example of where this is helpful. The Report object is declared in the Print Xtreme Orders form's Form_Load event. As such, the object stays loaded during the entire scope of the application. Every time the OK button is clicked, the application sets report properties based on the controls on the form and then either previews, prints, or exports the report.

The issue of clearing previous settings comes into play when a user runs the report more than once without ending and restarting the application. Because the Report object is never released and redeclared, all of its property settings remain intact when the user clicks OK subsequent times. If the VB code doesn't explicitly reset some of these previously set properties, the report may exhibit behavior from the previous time it was run that you don't expect. Executing the Report object's DiscardSavedData method clears many properties of settings from any previous activity, returning them to their state in the original .RPT file.

Simply execute the DiscardSavedData method for any Report objects that you want to "clean up," as shown in the following sample from the Xtreme Orders sample application:

```
Report.DiscardSavedData 'required for consistent results
```

This is particularly important if the .RPT file that you're integrating has been saved with File | Save Data with Report turned on. In this case, you'll probably want VB to discard the saved data at run time and refresh the report with new data from the database. In some cases, even if you do execute DiscardSavedData, a report with saved data will not behave as expected. This is still true in Crystal Reports 8. As a general rule, you'll want to save reports that will be integrated with VB without any saved data, to avoid any unexpected problems.

SQL Database Control

Many corporate databases are kept on client/server SQL database systems, such as Microsoft SQL Server, Oracle, and Informix. Thus, your VB applications will probably provide front-end interfaces to these database systems, and will need to handle SQL reporting, as well. The Automation Server contains several properties and methods that help when integrating reports based on SQL databases.

Logging On to SQL Databases

Your VB application probably already handles SQL database security, requiring the user to provide a valid logon ID and password at the beginning of the application. The user will be frustrated if the Automation Server asks for yet another logon when it comes time to open the report. By using methods and properties to pass logon information to the report, you can pass the valid ID and password the user has already provided, thereby allowing the report to print without prompting again.

One approach to passing logon information to a report is the Application object's LogOnServer method (this is one of the few times you will manipulate the Application object directly). This method logs on to a secure database and remains logged on until LogOffServer is executed or the entire application ends. Search Developer's Help for "LogOnServer method (Application object), Crystal Report Engine object library."

The other way to provide logon information from within your application is by executing the SetLogOnInfo method for the Report object's DatabaseTable object. If you have tables in your report that originate from different databases, this approach allows individual logon information to be provided for each table in the report. Search the online Help for "SetLogOnInfo method (DatabaseTable object), Crystal Report Engine object library."

Retrieving or Setting the SQL Query

When you submit a record-selection formula, as discussed earlier in the chapter, the Automation Server will generate a SQL statement to submit to the database server automatically. However, if the record-selection formula contains Crystal formulas, an OR operator, or other characteristics that prevent the Automation Server from including a WHERE clause in the SQL statement, report performance can be dramatically affected for the worse. You may, therefore, want to create the SQL statement the report uses directly in your VB application. As part of this process, you may find it helpful to retrieve the SQL statement that is being automatically generated.

The Report object provides a read/write SQLQueryString property that can be read or written to at run time. By examining the contents of this property, you can see what the SQL query is that the Automation Server is creating. You can then make necessary modifications to the query, based on the state of your application and the database that the report is based on. Then, pass the modified query back to the SQLQueryString property to be submitted to the server when the report runs.

Caution: As with Crystal Reports, you cannot modify the SELECT clause in the SQL query. Only FROM, WHERE, and ORDER BY clauses can be modified. Don't forget that you must still include the SELECT clause, however, and that it must not be changed from the original clause the Automation Server created.
Also, if you create an ORDER BY clause, you must separate it from the end of the WHERE clause with a carriage return/line feed character sequence. Use the vbCrLF VB constant to add the CR/LF sequence inside the SQL query.

Reading or Setting Stored Procedure Parameters

If your report is based on a parameterized SQL stored procedure, you will probably want to supply parameter values to the stored procedure from within your code, much as you will want to respond to any parameter fields that exist in the .RPT file.

Unlike the RDC, the Automation Server treats stored procedure parameters separately from report parameter fields. Stored procedure parameters are contained in their own DatabaseParameters collection contained within the Database object, below the Report object. The DatabaseParameters collection contains one DatabaseParameter object for each stored procedure parameter in the database. You can view or set the value of the parameter by manipulating the Value property of the DatabaseParameter object.

Note that you must always pass a string expression or variable to the Value property, even if the stored procedure parameter is defined as another data type—just make sure to pass a string that can be properly converted. The Automation Server will convert the parameter when it passes it to the database. For more information, search Developer's Help for "DatabaseParameter object (Crystal Report Engine object library)."

Automation Server Subreports

Manipulating subreports with the Automation Server is still a fairly straightforward process in Crystal Reports 8. To completely understand how subreports will interact with your VB code, consider that the main, or *containing,* report can contain a variable number of subreports. A subreport is actually considered to be a whole separate report, including its own sections, objects, record-selection formula, and other typical report properties. Manipulating each subreport piece is fairly easy.

To manipulate subreport properties, such as the record-selection formula, section or area formatting, or other settings discussed previously in the chapter, you need to assign the subreport to its own Report object variable. This is accomplished with the main Report object's OpenSubreport method. This method only takes one string argument, the name of the subreport to open. Consider the following sample code:

```
Dim mainReport as CRPEAuto.Report
Dim subReport as CRPEAuto.Report
…
Set mainReport = Application.OpenReport("C:\Samples\Subreport.RPT")
Set subReport = mainReport.OpenSubreport("Top 5 Products")
…
'Set selection formula for main report
mainReport.RecordSelectionFormula = "{Customer.Country} = 'USA'"
'Set selection formula for subreport
subReport.RecordSelectionFormula = "{Orders.Order Amount} < 1000"
```

Notice that two Report objects are declared: one for the main report, and one for the subreport. The main Report object variable is assigned to the containing report with the Application object's OpenReport method. Once the main report has been assigned to an object variable, that object’s OpenSubreport method will assign the subreport to the Subreport object variable. You can then manipulate the same properties and methods for the Subreport object that have been discussed earlier in the chapter.