The Report Engine API

Of the different report integration options available to Visual Basic, both the Crystal ActiveX control and the Automation Server eventually pass their calls to a lower-level base component known as the Crystal Report Engine API (REAPI). In fact, the Report Designer component (RDC) is the only interface that doesn't ultimately pass calls to the REAPI. If your development tool doesn't support these other methods, or you have other reasons for preferring API-level coding, you can program directly to this REAPI.

Although the term *API,* or *application programming interface,* is often used to refer to any number of ways of interfacing different software components, it originally referred to calling Windows dynamic link library (DLL) files from within a Windows programming language. Crystal Reports uses the term API in this fashion. The REAPI is comprised of one or more Windows DLL files. You can call functions from these DLL files from within your Visual Basic program.

With earlier versions of Crystal Reports, the REAPI was the only developer interface that existed. But, as Microsoft has refined its Component Object Model (COM) specifications, more and more COM automation server interfaces have surfaced as standards for development. Seagate makes clear in its Version 8 documentation that the RDC is the developer interface that will be the focus of its attention as the product progresses. While Seagate has upgraded the REAPI in Version 8 to include many new reporting features, it is the first time that some features are missing from this interface. In particular, the new Report Creation routines that allow you to create your own .RPT files entirely within code, are absent from this interface. You must use the RDC, discussed in Chapter 23, to accomplish this.

Using the REAPI has some other disadvantages, as well. While Windows DLL programming is the most mature method of Windows integration, it also tends to be the most complex. As you work through the sample Xtreme Orders application, you'll notice that it is quite a bit more code-intensive than applications created with other integration methods. Also, because the REAPI was originally designed with C programmers in mind, you'll find some functionality that is difficult, if not impossible, to implement in Visual Basic. Seagate includes a

REAPI "wrapper" DLL to solve some of these problems. But others, such as the ability to trap preview-window events, aren't possible with Visual Basic because of inherent limitations, such as the lack of pointer handling and the inability to call structures from within other structures.

This discussion of the REAPI may be helpful if you need to perform maintenance on existing VB programs that use the REAPI. If you are creating a new application, strongly consider using the RDC to integrate it with your application. If you are using a development tool that doesn't support a COM Automation Server, you may have little choice but to use the REAPI. In this event, you need to consider syntax conversion from the VB examples here to the particular language you are using.

Tip: All API calls exposed by the REAPI are documented in Developer's Help. Look for the DEVELOPR.HLP file in *Crystal Reports Program Directory*\Developer Files\Help.

Adding the Report Engine API to Your Project

The REAPI consists of a single base-level DLL file, CRPE32.DLL (Crystal Reports Print Engine 32-bit). Because this is a 32-bit DLL, it works only with 32-bit environments, such as Visual Basic 5 and 6 (earlier versions of Crystal Reports included a 16-bit DLL, but that is gone in Version 8). Also, don't forget that, unlike previous versions, this DLL is included only with the Developer Edition—it's not available with the Standard and Professional Editions of Crystal Reports.

Note: As with other integration methods, Seagate Software grants you a license to redistribute CRPE32.DLL, as well as other support DLLs, with your application royalty-free. You may copy these files along with your application files and distribute them to your customers or users at no additional charge. Chapter 26 discusses distribution issues in more detail.

You access REAPI features with VB function calls, much as you would access your own Sub or Function calls. These calls are predefined to point to CRPE32.DLL with VB DECLARE statements. While you could write your own DECLARE statements to call only the REAPI functions you are particularly interested in, Seagate makes life much easier for you by

predeclaring REAPI calls in an included .BAS module. By merely adding GLOBAL32.BAS to your application, all REAPI calls will be available globally throughout your application.

How you add a .BAS file to your project varies by VB version. For VB 6, you can choose Project | Add Module or Project | Add File from the pull-down menus. You can also right-click in the Project Explorer and choose Add | Module or Add | Add File from the pop-up menu. Navigate to the *Crystal Reports program directory*\Developer Files\include and locate GLOBAL32.BAS. When you add the module to your project, it appears in the Project Explorer under the Modules folder. If you look at the file, you'll see the declaration of constants, structures, and functions that point to the REAPI CRPE32.DLL file.

You'll probably want to add additional modules to your project, as well, depending on the functions you're planning on using in your application. As you look through Developer's Help at various API calls, you'll notice many PE (for "Print Engine") constants that are used to provide various parameter values. These are defined in the GLOBAL module along with REAPI calls.

There are other constants that are not defined in this module, though. In particular, you'll soon find examples of CW_USEDEFAULT in Developer's Help, as well as other generic Windows constants that are used in many API projects, not just those using Crystal Reports. These must be obtained from the Microsoft-provided WIN32API.TXT file that's installed with Visual Basic. You won't want to just add the entire file to your project. It's too large and contains many constants and procedures that you won't want. Just load the file into a text editor or word processor and copy selected constant declarations into your project. You can also use the API Viewer application to search WIN32API.TXT for just the constant declarations you need. Then, copy these constants to the Clipboard and paste them into a module in your VB application. In the Xtreme Orders sample application, only the necessary constants have been added to the winCONST module.

Tip: WIN32API.TXT is put in various locations, depending on the version of VB you're using. With VB 6, you'll find it in the \WINAPI subdirectory below the main Visual Basic directory. Launch the API Viewer by using the VB Add-In Manager from the Add-Ins menu.

And, finally, because of the inherent limitations of Visual Basic mentioned earlier in the chapter, Seagate has created "wrapper" .DLL and .BAS files that circumvent some of these limitations. In particular, the VB wrapper helps in setting parameter fields, exporting to e-mail and various file formats, and logging on to SQL databases. If you need any of this functionality in your application, add the wrapper to your project. However, you may not need to use the VB wrapper, depending on the functions you use in your VB application. If you begin your project without it and later need some of its functions, you can simply add it at that time.

The wrapper DLL is CRWRAP32.DLL (for 32-bit environments). Declarations to the DLL are provided by Seagate in CRWRAP.BAS, located along with GLOBAL32.BAS in *Crystal Reports program directory*\Developer Files\include. Adding this module to your project will declare additional constants and API calls specifically for Visual Basic.

After you add all the necessary files to your project, you're ready to begin coding. Figure 1 shows the VB IDE for the Xtreme Orders sample application. Notice the three different modules that have been added, along with some declarations in GLOBAL32.BAS.

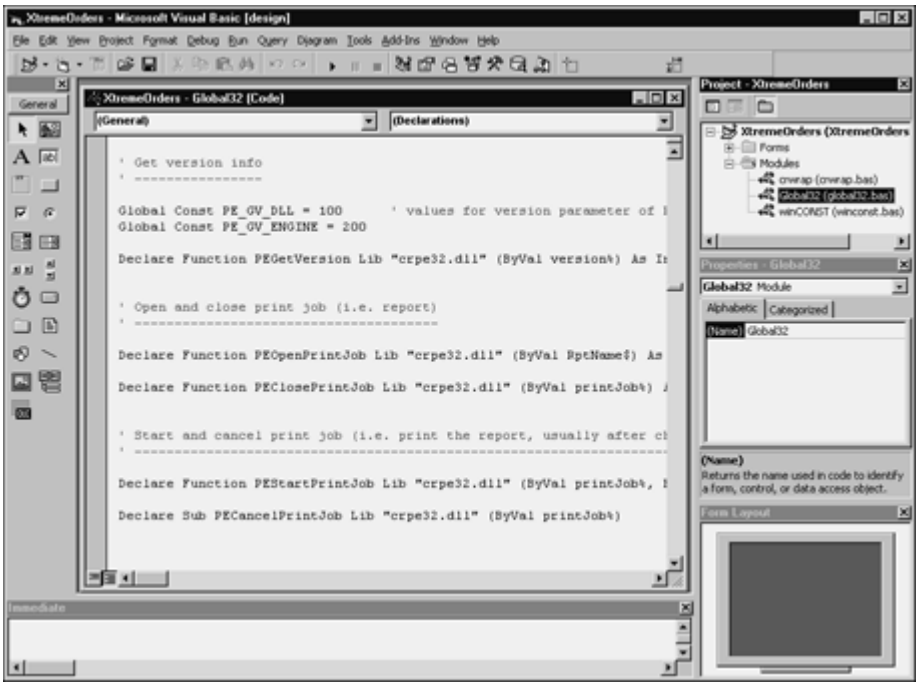

Figure 1. Modules added to Xtreme Orders application.

Creating a Simple Application with the REAPI

Once the REAPI has been added to your project, you're ready to begin coding your application. Coding with the REAPI is accomplished by calling the REAPI functions defined in the modules you just added. REAPI functions coming from GLOBAL32.BAS begin with the letters PE (indicating Print Engine). Functions coming from the CRWRAP.BAS wrapper module begin with the letters crPE.

There are six basic steps to integrating a report with the REAPI:

1. Initialize the Crystal Report Engine (PEOpenEngine).
2. Open a specific report (PEOpenPrintJob).
3. Choose an output destination (PEOutputToWindow, PEOutputToPrinter, or various crPEExport calls).
4. Process the report (PEStartPrintJob).
5. Close the specific report that was opened earlier (PEClosePrintJob).
6. Close the Crystal Report Engine at the end of your application (PECloseEngine).

Based on program logic, you may repeat several of these steps multiple times, depending on how many reports you are integrating in your application at once, how many times they may be processed, and so on. Also, you'll probably make many additional calls between PEOpenPrintJob and PEStartPrintJob to customize report features, such as the preview window, record-selection criteria, formulas and parameter fields, and report-section formatting.

At the most basic level, you can create an application to preview a report in the preview window with no customization, by using the following lines of code:

```
Dim intJob As Integer   'for report job number
Dim intResult As Integer   'to read results of PE calls
Private Sub Form_Load()
   'Open the print engine
   intResult = PEOpenEngine()
   'Open the report
   intJob = PEOpenPrintJob(App.Path & "\Xtreme Orders.RPT")
End Sub
```

```
...
...
Private Sub cmdOK_Click()
   'Send to preview window
   intResult = PEOutputToWindow(intJob, "Xtreme Orders Report", _
      0, 0, 0, 0, 0, 0)
   'Process the report
   intResult = PEStartPrintJob(intJob, True)
End Sub
...
...
Private Sub Form_Unload(Cancel As Integer)
   intResult = PEClosePrintJob(intJob)
   PECloseEngine
End Sub
```

Consider these points about this sample code:

- Most PE calls return integer results (with the exception of PECloseEngine). As a general rule, a result of 0 indicates that the call failed in some fashion. For most calls, a result of 1 indicates success. For PEOpenPrintJob, a 0 indicates failure and any other integer is the job number that is used in subsequent calls that relate to that particular print job. This sample code contains no error-handling logic. Error handling is discussed later in the chapter.
- You'll probably want to open and close the print engine as few times as possible during your application. In this case, there's only one form in the application, so the engine is opened in Form_Load and is closed in Form_Unload. In other situations, you may want to do this in the Load and Unload events for the main form in your report, or perhaps in Sub Main( ).
- If you want to have more than one report open at one time, you may call PEOpenPrintJob as many times as necessary. Each open report will then have a unique job number to use in subsequent Print Engine calls. If you are simply going to have one report open at one time, but perhaps use several different reports throughout the duration of your application, you can call PEOpenPrintJob and PEClosePrintJob every time you wish to change reports.
- PEStartPrintJob includes a second parameter that's set to true. This is the only value you can supply to this parameter, so always submit true to this call.

Controlling the Preview Window

In the sample code shown previously, very few customized options were provided to the PEOutputToWindow call. The only customization done to the preview window at all was setting the window title. The remaining function parameters were set to zeros to specify default behavior. These other parameters relate to the size and position of the preview window. You'll probably want to further customize preview-window behavior, however. Perhaps you want to enable drill-down with a summary report, or eliminate the export or print buttons so that a user can't print or export the report.

The REAPI allows you to customize all of this preview-window behavior by modifying members of the PEWindowOptions structure. This structure is one of several defined by GLOBAL32.BAS to control a series of related settings as a single group. These structures contain several elements that can be set to specific values, or left unchanged by setting them to the predefined constant PE_UNCHANGED. Examine the following sample code:

```
'Set preview window options
Dim WindowOptions As PEWindowOptions
...
WindowOptions.StructSize = Len(WindowOptions)
' required to avoid error 538
With WindowOptions
   .hasGroupTree = 1
   .canDrillDown = 1
   .hasNavigationControls = PE_UNCHANGED
   .hasCancelButton = PE_UNCHANGED
   .hasPrintButton = PE_UNCHANGED
   .hasExportButton = PE_UNCHANGED
   .hasZoomControl = PE_UNCHANGED
   .hasCloseButton = PE_UNCHANGED
   .hasProgressControls = PE_UNCHANGED
   .hasSearchButton = PE_UNCHANGED
   .hasPrintSetupButton = PE_UNCHANGED
   .hasRefreshButton = PE_UNCHANGED
End With    'WindowOptions
intResult = PESetWindowOptions(intJob, WindowOptions)
```

Notice that you must declare a variable to hold the PEWindowOptions structure elements. The other requirement is to set the StructSize member to the size of the whole structure. This can be accomplished by just using Len of the structure variable itself. Then, set each of the structure elements to 1 to enable the option, or 0 to disable the option. Make sure you specifically set each structure member that you want to use. If you don't set an option, the option will be disabled. By using PE_UNCHANGED, the option will exhibit default behavior or stay the same as it was set earlier in your program. Once you've assigned all the options, pass the structure variable, along with the job number, to the PESetWindowOptions call to actually set the preview-window options.

Note: Although the REAPI facilitates trapping preview-window events, such as drill-down and button clicks, this requires complex Windows programming. Because of inherent Visual Basic limitations, you cannot develop these routines using the REAPI. If you need to trap these events, you need to use the RDC to integrate your report.

Controlling Record Selection

Controlling report record selection is an integral part of VB report integration. You'll typically use your VB user interface to gather information from the user that will make up the record-selection criteria for the report. Then, you make the proper REAPI call to set the report's record-selection formula.

Because you must pass a valid Crystal Reports record-selection formula to the REAPI, you need to be familiar with the Crystal Reports formula language in general, and record-selection formula concepts in particular. 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 limits 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 PESetSelectionFormula call to pass a record-selection formula to the report. The call takes two parameters: the job number of the report, and a string variable or expression that comprises the record-selection formula. The following code from the sample application sets record selection, based on the contents of the From Date and To Date text boxes. Notice that a work-string variable is being used to hold the formula before it's passed to the REAPI call. This allows you to stop program execution and look at the contents of the string to make sure it contains a syntactically correct formula.

```
'Supply record selection based on dates
strWork = "{Orders.Order Date} in #" & _
   txtFromDate & "# to #" & txtToDate & "#"
intResult = PESetSelectionFormula(intJob, strWork)
```

Remember that the string value you pass must adhere *exactly* to the Crystal Reports formula syntax. This includes correct use of Crystal reserved words and punctuation. The example from the sample application shows the necessity of adding pound signs (#) to indicate that the values being passed to the In function are dates. Also, don't forget the required quotation marks or apostrophes around literals that are used in comparisons.

There are also important considerations if you are using a SQL database. The REAPI 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, typically causes record selection to be moved to the client (the PC) instead of to the database server, resulting in reduced 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 the report will use right in your VB application and submit it to the report by using the separate REAPI PESetSQLQuery call.

Setting Formulas

You'll also want to use Visual Basic to change the contents of Crystal Report formulas at run time. This is necessary for changing formulas that may be related to groups that are also being changed, for changing textual information on the report, or for using the VB user interface to change a report calculation.

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

Use the REAPI PESetFormula call to change an existing formula (you can't create new formulas that don't already exist in the report). The call accepts three parameters: the job number of the report you wish to modify, a string variable or expression that contains the name of the formula you want to change (without the @ sign), and a string variable or expression that contains the new text for the formula.

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
   strWork = "{Orders.Order Amount}"
Else
   strWork = "{Orders.Order Amount} *" & Str(txtTaxRate / 100 + 1)
End If  'txtTaxRate = ""
intResult = PESetFormula(intJob, "Order + Tax", strWork)
```

The string passed with the third parameter must correctly conform to the Crystal Reports formula syntax. Here, the Order + Tax formula is defined in Crystal Reports as a numeric formula. Therefore, 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 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.

You may also need to change simple string formulas in your program. The sample application uses the @Sub Heading string formula in the page header to print a description of the report options 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
strWork = "'" & txtFromDate & " through " & txtToDate
strWork = strWork & ", By " & cboGroupBy
If txtTaxRate = "" Then
   strWork = strWork & ", No Tax'"
Else
   strWork = strWork & ", Sales Tax = " & txtTaxRate & "%'"
End If  'txtTaxRate = ""
intResult = PESetFormula(intJob, "Sub Heading", strWork)
```

Passing Parameter Field Values

Your report may contain parameter fields to prompt the user for information whenever the report is refreshed. If the parameter fields are ignored in the VB application, the REAPI automatically prompts the user for them when the report runs. To take advantage of your VB user interface, you'll want to be able to supply these parameter field values from within your code. Setting parameter-field values in code is particularly helpful if you're integrating a predesigned report that is heavily based on parameter fields, or if you're sharing a report on a network with other non-VB users who will need to be prompted when they run the report directly in Crystal Reports.

There are several ways of setting parameter-field values with the REAPI. The PESetNthParameterField or PEAddParameterCurrentValue calls (defined in GLOBAL32.BAS) accept several parameters, including the PEParameterFieldInfo structure. Seagate has also provided the crPESetNthParameterField call in the REAPI wrapper (defined in CRWRAP.BAS), which doesn't require the structure definition. Although the wrapper-defined function is easier to

use, it does not work properly with the initial release of Crystal Reports 8—you need to use an alternate PE method.

PEAddParameterCurrentValue uses the PEValueInfo structure, which is declared in GLOBAL32.BAS. You need to Dim a structure variable for this structure and then set various structure member values to control the parameter field you want to set. Then, include the structure variable in the PEAddParameterCurrentValue call. A particular structure member you'll want to pay attention to are StructSize, which must be set to the actual size of the structure itself by using Len(*structure variable*). You'll also want to declare the data type of the parameter field by setting the valueType member to one of the predefined constants indicating "VI" value types. Then, there are differing structure members (a different member for each data type) that will ultimately contain the value you wish to pass to the parameter field. In the following sample code, the value being passed is a number, so the actual value to be given to the parameter field will be placed in the structure's viNumber member.

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 for correctly passing the value to the parameter field:

```
Dim ValueInfo As PEValueInfo 'parameter field options structure

...

'Set ?Highlight parameter value
'Note: crPESetNthParameterField doesn't require the structure definition,
'      but doesn't work properly in the version 8 CRWRAP32.DLL

ValueInfo.StructSize = Len(ValueInfo) ' Set structure size
ValueInfo.valueType = PE_VI_NUMBER  ' value passed will be a number
If txtHighlight = "" Then
    ValueInfo.viNumber = 0
Else
    ValueInfo.viNumber = CDbl(txtHighlight)
End If  'txtHighlight = ""
intResult = PEAddParameterCurrentValue(intJob, "Highlight" & Chr$(0), _
            "" & Chr$(0), ValueInfo)
```

Notice that if the user leaves the Highlight text box blank, you still need to pass a numeric value to the parameter field, even if it's just 0. Also, since the parameter field is being defined with a numeric data type, the CDbl function is being used to ensure that the contents of txtHighlight are submitted as a double-precision numeric value. Also notice that string values passed to PE calls (such as the parameter field name) must be null-terminated. This is accomplished by adding the Chr$(0) function to the string.

Note: New Crystal Reports 8 parameter-field features, such as range values and multiple values, are exposed by the REAPI, but not by the VB wrapper. Some of this new functionality may not be fully available because of Visual Basic limitations. If your application depends on this functionality, you may want to consider integrating the report with the RDC.

Manipulating Report Groups

You can add significant flexibility to your application by changing report grouping at run time. In many cases, you can actually make it appear that several different reports are available for users to choose from. In fact, the application will be using the same Report object, but grouping will be changed at run time from within your application.

The Xtreme Orders sample application features 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 change the field the first report group is based on, as well as choose quarterly grouping if the Order Date field is chosen. To familiarize yourself with various grouping options, refer to Chapter 3.

The REAPI provides the PESetGroupCondition call to change the grouping options for any group on your report. Although the call only requires you to supply five parameters, you need to use several other built-in REAPI calls within PESetGroupCondition to get the proper results. Look at the sample code from the Xtreme Orders report:

```
'Set grouping
If cboGroupBy = "Quarter" Then
   intResult = PESetGroupCondition(intJob, _
   PE_SECTION_CODE(PE_SECT_GROUP_HEADER, 0, 0), _
```

```
   "{Orders.Order Date}", PE_GC_QUARTERLY, PE_SF_ASCENDING)
Else
   intResult = PESetGroupCondition(intJob, _
   PE_SECTION_CODE(PE_SECT_GROUP_HEADER, 0, 0), _
   "{Customer.Customer Name}", PE_GC_ANYCHANGE, PE_SF_ASCENDING)
End If  'cboGroupBy = "Quarter"
```

You'll notice the first parameter is the now-familiar report job number. However, indicating which group you actually want to change with the second parameter requires some extra work. Ultimately, the REAPI expects a number indicating which section of the report contains the group you want to change. By using the built-in PE_SECTION_CODE function defined in GLOBAL32.BAS, you can supply section information in a more logical fashion and have the correct number returned.

The PE_SECTION_CODE function itself takes three numeric parameters: section type, group number, and section number. The section type number can be supplied with a predefined constant, such as PE_SECT_GROUP_HEADER, indicating the area you wish to modify. If this area is a group header or footer, you can indicate which group you wish to modify with the zero-based second parameter (0 is group one, 1 is group two, and so on). If the section is not a group, simply pass 0 to this argument. The third argument, also zero-based, is used if there is more than one section in the area. For example, if you have created Details sections a, b, and c, 0 refers to Details a, 1 to Details b, and 2 to Details c.

Caution: To get a list of section code constants provided by GLOBAL32.BAS, search Developer's Help for "Encoding."

The remaining three parameters in PESetGroupCondition are a string variable or expression indicating the database or formula field that you want the group based on, the condition that will cause a new group to appear (when using Date, Time, or Boolean grouping), and the order in which you wish groups to appear. The last two parameters are best supplied with predefined constants supplied by GLOBAL32.BAS. The PESetGroupCondition topic in Developer's Help provides the available constants.

Conditional Formatting and Formatting Sections

If your VB application displays a report in the preview window, you'll be able to take advantage of interactive reporting features, such as the group tree and drill-down. In particular, drill-down capabilities allow a report to be presented initially at a very high summary level, showing just subtotals or grand totals, or perhaps just showing 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 drill-down are limited only by the number of report groups you create on your report.

The Xtreme Orders sample application Summary Report Only check box lets the user choose whether or not to see just a summary report. When this check box is selected, you need to hide the report's Details section with VB code so that only group subtotals appear on the report. In addition, the code has to control the appearance of the report's two Page Header sections (Page Header a and Page Header b), as well as two Group Header #1 sections (Group Header #1a and Group Header #1b).

The page header and group header manipulation will accommodate two different sections of field titles that should 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 want the field titles to appear in the Group Header section of a drill-down tab only 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 will indicate which group the following set of orders applies to. However, if the report is showing only summary data, showing both the group header and group footer would be repetitive—the group footer already contains the group name, so showing the group header as well would look 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 provides 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 Report Types

You'll 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). A somewhat obscure relationship among areas and sections affects 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.

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.

There are two REAPI calls that accomplish the required formatting. First is PESetAreaFormat, which sets formatting options for an area in the report. An *area* encompasses

one or more related sections. For example, the page header is considered an individual area. However, both Group Header #1a and Group Header #1b are combined in one single Group Header #1 area.

The REAPI also exposes PESetSectionFormat, which formats a section in the report. A *section* encompasses each individual section of the report, regardless of how many 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.

Both calls use the PESectionOptions structure defined in GLOBAL32.BAS. This structure contains members that allow you to control whether an area is shown or hidden for drill-down, whether a section is suppressed or visible, what the background color of the section is, and so forth. You must declare a variable that points to this structure, and then set the individual options of the structure for the variable. Then, pass the variable to PESetAreaFormat and PESetSectionFormat. The complete syntax for these calls, as well as descriptions for all members of the structure, can be found in Developer's Help.

Keeping in mind the previous discussion and the hide/suppress requirements outlined in Table 1, look at the code from the sample application that follows:

```
Dim SectionOptions As PESectionOptions 'report section options structure
...
'Hide/show areas and sections
SectionOptions.StructSize = Len(SectionOptions)
If chkSummary Then
   SectionOptions.showArea = 0 'Hide for drill down
   SectionOptions.Visible = 1 'Don't suppress
   SectionOptions.backgroundColor = PE_UNCHANGED
      'Set to avoid black bkgrnd

   'set for Details area
   intResult = PESetAreaFormat(intJob, _
   PE_SECTION_CODE(PE_SECT_DETAIL, 0, 0), SectionOptions)

   'set for all of Group Header 1 area
   intResult = PESetAreaFormat(intJob, _
   PE_SECTION_CODE(PE_SECT_GROUP_HEADER, 0, 0), SectionOptions)
```

```
   'set for Group Header 1b section
   intResult = PESetSectionFormat(intJob, _
   PE_SECTION_CODE(PE_SECT_GROUP_HEADER, 0, 1), SectionOptions)

   'set for Page Header b section
   SectionOptions.Visible = 0 'Suppress
   intResult = PESetSectionFormat(intJob, _
   PE_SECTION_CODE(PE_SECT_PAGE_HEADER, 0, 1), SectionOptions)
Else
   SectionOptions.showArea = 1 'Show
   SectionOptions.Visible = 1 'Show
   SectionOptions.backgroundColor = PE_UNCHANGED
      'Set to avoid black bkgrnd

   'set for Details area
   intResult = PESetAreaFormat(intJob, _
   PE_SECTION_CODE(PE_SECT_DETAIL, 0, 0), SectionOptions)

   'set for all of Group Header 1 area
   intResult = PESetAreaFormat(intJob, _
   PE_SECTION_CODE(PE_SECT_GROUP_HEADER, 0, 0), SectionOptions)

   'set for Page Header b section
   intResult = PESetSectionFormat(intJob, _
   PE_SECTION_CODE(PE_SECT_PAGE_HEADER, 0, 1), SectionOptions)

   'set for Group Header 1b section
   SectionOptions.Visible = 0
   intResult = PESetSectionFormat(intJob, _
   PE_SECTION_CODE(PE_SECT_GROUP_HEADER, 0, 1), SectionOptions)
End If  'chkSummary
```

Notice that several calls are made using the same SectionOptions structure variable. If the settings in the SectionOptions structure variable are the same for multiple sections or areas, you can simply call PESetAreaFormat or PESetSectionFormat multiple times with the same variable. If you need to change options between calls, only the structure members that require modification need to be changed—the others can be left alone. However, it's crucial to set as many members

as necessary to ensure the correct report behavior. Even though the report sections have the default background color set in the .RPT file, you must specifically supply the PE_UNCHANGED constant to indicate no change. Otherwise, the background color will show up in black!

Notice also that the same PE_SECTION_CODE section- and area-encoding function that was described earlier in the chapter, under "Manipulating Report Groups," is used here, as well. By supplying the three parameters to PE_SECTION_CODE, you can determine which areas and sections are to be formatted. Remember that the indexes for this function are zero-based: 0 indicates the first (or only) Details section, whereas 1 indicates the *second* Page Header section.

Caution: Members of the two structures used in the Xtreme Orders sample application (PEWindowOptions and PESectionOptions) require only numeric values. Some REAPI structures accept string values, as well. The REAPI requires you to "null-terminate" any string values you pass to these structures. For example, if you're setting the fieldName member of a PEGroupOptions structure variable, you should terminate the value with a null character. Instead of

```
GroupOptions.fieldName = "{Orders.Order Date}"
```

use

```
GroupOptions.fieldName = "{Orders.Order Date}" & Chr$(0)
```

Choosing Output Destinations

The first method of output, discussed at the beginning of the chapter, is the preview window. The REAPI also supports 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 typed into a text box. Once this selection has been made, you need to set the output destination automatically in your VB code, based on the user's selection.

The REAPI provides different calls for different report destinations. Make the correct call based on the user's choice, including setting options for the particular output method chosen. PEOutputToPrinter will send the report to a printer, and PEExportTo will export the report to a data file or attach a file to an e-mail message. Once these options have been set, PEStartPrintJob will send the report to the selected output location.

Although you can use PEExportTo to choose any file or e-mail output destination, it requires setting up several different structures and options. Seagate has provided more intuitive calls in the CRWRAP.BAS wrapper definitions. In particular, crPEExportToMapi can attach a report to an e-mail message simply by supplying parameters—no structure definitions are required.

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:

```
'Set output destination
If optPreview Then
   intResult = PEOutputToWindow(intJob, "Xtreme Orders Report", _
   0, 0, 0, 0, 0, 0)
End If  'optPreview
If OptPrint Then
   intResult = PEOutputToPrinter(intJob, 1)
   'will print to printer assigned w/report or Windows default
End If 'OptPrint
If OptEmail Then
   intResult = crPEExportToMapi(intJob, txtAddress, "", _
   "Here's the Xtreme Orders Report", _
"Attached is a Word document showing the latest Xtreme Orders Report.", _
   "u2fwordw.dll", crUXFWordWinType, 0, 0, "", "")
End If  'optEmail
```

Note: For the sake of brevity, code to set preview-window options has been eliminated from this sample. Look earlier in the chapter for details on setting preview-window options.

If you wish to send the report to a printer, use PEOutputToPrinter. This call accepts only two parameters: the job number of the report to print, and the number of copies of the report to

print. The report will be sent to the printer that was chosen when the report was saved, or to the Windows default printer if no printer is saved with the report.

Unlike other integration methods, the REAPI doesn't provide a simple option to display the Printer Options dialog box before printing. While the REAPI does provide the PESelectPrinter call to choose a print destination, it requires passing a Windows DEVMODE structure that contains print options. This structure, in combination with the Printer Options dialog box from Windows' common dialog functions, can be used to display a dialog box and pass printer options to PESelectPrinter.

As opposed to the REAPI's PEExportTo call, which exports to any destination with one call, the VB wrapper DLL exposes separate crPEExportTo calls for each destination. The crPEExportToMapi call is exposed by the wrapper to export to a MAPI e-mail system. The call requires no structure manipulation and allows all required options to be passed as parameters. While a description of all available parameters is best left to Developer's Help, several parameters deserve particular attention, such as the To and CC addresses, the e-mail subject line, and e-mail message text. These are all passed as simple string variables or expressions.

Because Crystal Reports always sends e-mail reports as attached files, you need to specify the format of the file that will be attached. The Xtreme Orders sample application will attach a Word document to the e-mail message. Two parameters make this determination. FormatDLLName (the sixth parameter) supplies crPEExportToMapi with the name of the Crystal Reports export .DLL file that will convert the report. Developer's Help contains a list of all available .DLL files. FormatType (the seventh parameter) accepts a predefined constant indicating the type of export that will be created. While this may seem redundant, considering the FormatDLLName parameter, a single format .DLL file can actually export to several different formats. For example, the U2FXLS.DLL format DLL can export to several versions of Excel.

Note: If you use crPEExportToMapi to attach the report to an e-mail message, the user's PC needs to have a MAPI-compliant e-mail client, such as Microsoft Outlook or Windows Messaging, installed and configured.

Error Handling

In normal report-integration programming, you may encounter unexpected errors from the REAPI, and the somewhat verbose coding requirements of this integration method may exacerbate this problem. Unlike other integration methods, however, the REAPI doesn't throw standard VB errors that can be trapped with On Error Goto. Instead, each PE or crPE call returns an integer indicating whether the call succeeded or failed (1 means success, 0 means failure). You can check the return code to determine the success or failure of the call, as follows:

```
If intResult <> 1 Then
   HandleError
   Exit Sub
End If  'intResult <> 1
```

Note: You'll notice that HandleError is not called with any parameters and does not actually return any results. Because it resides in the same form where intJob is declared, it can determine the report that caused the error. Typically, you will want to pass the job number so that it can be used in the error-handling routine. You can also choose to have the routine return the error string.

While you can obviously design your code to handle REAPI errors as you see fit, it may be easy to create a single point of error handling. The Xtreme Orders sample application simply contains the HandleError subroutine in the main application form to deal with REAPI errors. If your application contains more than a single form, you may wish to create a centralized error handler in a module, or different handlers for each form.

In any event, you have to conduct some further coding "investigation" to determine what error actually resulted in a REAPI call failure, because the return code indicates only that the call *did* fail. There are several additional REAPI calls to help determine the cause. PEGetErrorCode is the simplest. It only needs one parameter: the report job number (just supply 0 if a job hasn't been opened yet). It will return an integer indicating the REAPI error code of the most recent error. (Search Developer's Help for a complete listing of REAPI error codes.)

While this may be sufficient for your application, you may want to provide more descriptive error messages to your users, instead of just numbers, in case an unanticipated report

error occurs. Although you conceivably could build your own lookup table and type in all the Crystal Reports error messages yourself, the REAPI provides several calls to retrieve a descriptive error message to be used along with, or in place of, the error code. PEGetErrorText allows you to retrieve the actual error message of the most recent error, instead of just the error code. The complication is that PEGetErrorText doesn't return an actual string—it returns only a *handle* to a string. You then need to pass the string handle to another call that will actually get the error message in a string variable. Because the regular PEGetHandleString function is difficult to implement in Visual Basic, Seagate has defined the alternative crvbHandleToBStr call in CRWRAP.BAS to simplify retrieving the error text.

Examine the following sample code from the Xtreme Orders sample application:

```
Private Sub HandleError()
Dim lngHandle As Long 'gets string handle
Dim intLength As Integer ' gets string length
Dim strReturnText As String 'gets actual string
Dim intError As Integer 'keep error code

intError = PEGetErrorCode(intJob)
Select Case intError
   Case 541
      MsgBox "Invalid e-mail address or other e-mail/export error", _
      vbCritical, "Xtreme Orders Report"
      txtAddress.SetFocus
   Case 545
      MsgBox "Report Canceled", vbInformation, "Xtreme Orders Report"
   Case Else
      'get handle and length
      intResult = PEGetErrorText(intJob, lngHandle, intLength)
      strReturnText = String$(intLength, 0) 'stuff string with nulls
      'NOTE: Failure to stuff string may result in Invalid Page Faults
      'get the string
      intResult = crvbHandleToBStr(strReturnText, lngHandle, intLength)
      MsgBox "Report Error:" & vbCrLf & intError & " - " & _
      strReturnText, vbCritical, "Xtreme Orders Report"
End Select 'Case intError
End Sub
```

Note that the error code is simply placed in intError for use in the Case statement and a message box. Based on this error code, e-mail-related errors or report cancellation is simply handled with appropriate message boxes.

If an unanticipated error occurs, extra steps are required to get the actual error string. You'll note that several variables have been declared to assist in this process. The lngHandle variable is used to actually contain the string handle returned by PEGetErrorText. The intLength variable is also used by PEGetErrorText to indicate the length of the error text. The actual error message itself is stored in strReturnText. Initially, PEGetErrorText is called with three parameters: the report job number, the handle variable, and the length variable. However, now you need to somehow turn the handle and the length into a meaningful string to be placed in strReturnText.

Before you actually make the call to fill the string, you should stuff the string variable with the same number of null characters as there are characters in the error message. The String$ function can be used with the intLength variable and 0 (to signify an ASCII zero, or null) to accomplish this. Seagate indicates in sample code that accompanies Crystal Reports that failure to do this may result in Invalid Page Fault errors.

Once the string has been stuffed, call the wrapper DLL's crvbHandleToBStr function to retrieve the actual contents of the error message. Supply three parameters: the string variable, the handle variable, and the length variable. The call will replace the nulls in the string variable with the actual error text, which can be used to display a meaningful error message.

Tip: This string-handling logic applies to many other REAPI calls that return strings. Keep it handy for future use!

Other REAPI Calls

While many of the most common REAPI calls necessary for report integration have been covered in this chapter, there are others that merit some discussion. Because many reports are based on client/server or other ODBC or SQL data sources, you need to know how to manipulate these types of databases with the REAPI. Also, if you will be integrating reports that contain subreports, you should know how to manipulate the subreports in the same way that you work

with the main report. And, to ensure that your integrated reports display the most current data from the database, you'll want to know how to discard any saved data that may be stored in the .RPT file.

Discarding Saved Data

As a general rule, saving data with an .RPT file that you will be integrating is a waste of disk space. While there may be situations in which your VB application will benefit from a Crystal Report with saved data, you'll most often want to have VB refresh the report with current database data. If the .RPT file is saved with File | Save Data with Report turned off, the .RPT file will be smaller in size and will always be refreshed with the most current data from the database.

If the .RPT file contains saved data, there is a REAPI call that will discard it and ensure that the report is displaying the most current set of data from the database. Use PEDiscardSavedData with one argument, the report job number:

```
intResult = PEDiscardSavedData(intJob)
```

SQL Database Control

Companies continue to convert mainframe or minicomputer applications to client/server platforms, as well as basing newer systems on SQL database technology. Your VB applications will probably provide front-end interfaces to these database systems, and will need to handle SQL reporting, as well. The REAPI exposes several calls that help when integrating reports based on SQL databases.

Logging On to SQL Databases

Because 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), you'll want to avoid requiring the user to log on to the report database separately. By using REAPI calls 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.

While the REAPI exposes several SQL database functions, the VB wrapper provides functions that require no structure manipulation. These functions can be used to provide logon information either for a database that will apply throughout the VB application, or for individual tables used in the report.

Both crPELogOnServer and crPESetNthTableLogonInfo are documented in Developer's Help. Parameters for the calls include the report job number, server name, user ID, password, and so forth. crPELogOnServer contains one argument that may require you to do a little research to properly provide it. The database .DLL name used to connect to the server must be provided as a string. These .DLL filenames follow the general format of P2B*.DLL for non-SQL databases, and P2S*.DLL for SQL databases.

Tip: If you're unsure what database driver your report is using, open the report in Crystal Reports and choose Database | Convert Database Driver. The database driver being used by the report will appear after the word "From" in the dialog box.

Retrieving or Setting the SQL Query

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

If you need to use the SQL query that the REAPI is automatically generating, use PEGetSQLQuery to get the query. As with PEGetErrorText, described under "Error Handling" earlier in the chapter, you need to use a pointer to the SQL query string along with crvbHandleToBStr to get the actual SQL query in a string. You can then modify the WHERE clause or other parts as necessary, and resubmit the query using PESetSQLQuery. If you already know what SQL statement you want to create, you can simply submit it with PESetSQLQuery—you don't have to get anything first.

Note: 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 REAPI 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 Visual Basic 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.

The REAPI exposes two calls for reading or setting stored procedure parameters. PEGetNthParam returns any existing value from the parameter. Arguments include the report job number, the zero-based index of the parameter (the first is 0, the second 1, and so on), and a text handle and length. Use previously discussed text-handling routines to retrieve the value.

If you just want to set the value of the stored procedure parameter, use PESetNthParam. Arguments include the report job number, the zero-based index of the parameter, and a string variable or expression containing the value you want to pass to the parameter.

Note: 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 REAPI will convert the parameter when it passes it to the database.

Report Engine API Subreports

Subreport handling in the REAPI is almost identical to handling the main report. 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 an entirely separate report, including its own sections, objects, record-selection formula, and other typical report properties.

Manipulating each subreport piece is essentially identical to working with the main report, except that subreports have different report job numbers. You obtain a new job number for the subreport by issuing a PEOpenSubreport call after the main report has been assigned a job number. Consider the following sample code:

```
Dim intJob As Integer  'job number for main report
Dim intSubJob As Integer   'job number for sub report
...
intJob = PEOpenPrintJob("Xtreme Orders w-Subreport.RPT")
intSubJob = PEOpenSubreport(intJob, "Top 5 Products Subreport")
```

You now have two job numbers: intJob refers to the main report, and intSubJob refers to the subreport named Top 5 Products Subreport, contained in the main report. You can now make any number of necessary REAPI calls and pass the desired report job number to the call, depending on whether you want the call to change the main report or the subreport. For example, to set selection criteria for both the main and subreports, you might use the following code:

```
intResult = PESetSelectionFormula(intJob, _
  "{Customer.Country} = 'USA'") 'main report
intResult = PESetSelectionFormula(intSubJob, _
  "{Customer.Country} = 'USA'") 'subreport
```