3

SCRIPT Engine

In the previous chapters we described in glorious details all the GUI tools that are contained in ELS-Script® package. In particular, we have seen that the *Report Designer* application is a profound tool for the creation and development of report scripts, and comes with a SCRIPT compiler and a design-time SCRIPT engine for debugging report scripts during their development. ELS-Script® package also contains two versions of the SCRIPT report engine, namely, the **GEV**-engine and the **E**-engine, both of which are currently available as COM components. The ELS-Script® package may optionally include a .NET compatible version of the **E**-engine, with built-in ADO.NET access, XML SCRIPT object, as well as other .NET technologies.

In this chapter, we will first outline the two versions of the SCRIPT engine, followed by an exposition of the architecture of the SCRIPT framework. Then we will describe all the API functions and members that are exposed in both engines, followed by illustrations of their usage via detailed sample host applications and report scripts.

In brief, we will discuss the following issues:

- Architecture of the SCRIPT framework
- SCRIPT engine types and usage configurations
- Utilizing the **GEV**-engine with the **Report Generator** and **Report Viewer** windows
- Integrating SCRIPT engine capabilities entirely in a server-based ASP application
- Outlining all the API functions, properties and events of SCRIPT engines
- Illustrating the API usage with detailed sample applications and report scripts

Architecture of SCRIPT Framework

The SCRIPT framework consists of several modules corresponding to various stages in the cycle of report creation and integration. In particular, these stages may involve the creation and design of report scripts, the diagnosis and compilation of these report scripts into binary report files, the integration of the run-time SCRIPT engine with host application, and the generation of report output at run-time.

Refining these stages a little further, the framework may be organized into the following parts:

- Data access and retrieval definition tools,
- DHTML design and edit tools,
- Data row tabulation, group and summary definition tools,
- Data field and expression insertion and format definition tools,
- Query form design and parameter option definition tools,

- SCRIPT edit and manipulation tools,
- Report script compilation and diagnostic tools,
- Binary report builder and optionally report catalog tool,
- SCRIPT engines,
- Configuration and integration of the SCRIPT engine via API functions, properties and events.

Report Design and Compilation

The report creation and design, essentially begins with the user selecting a standard template from the library of *Standard Report Templates*, which creates a draft template *ELS*-file containing script, copied from the selected standard template. Recall that an *ELS*-file is a report script file composed of DHTML, ASP, Java or VB Script segments intertwined with SCRIPT language code. Once such an initial template is created, the user may utilize various tools available in the *Report Designer* application to edit and design the report script.

The connection manager is used to define data access to the data sources, while the query and data shape builders are used to define SQL or Shape commands to be inserted into the report script. Similarly, XML data may be retrieved via XML objects. Once data sources and XML objects are added to the report script, the data fields or XML fields may be inserted via the **Data Fields** tool. Moreover, expressions derived from these data fields and SCRIPT functions may also be inserted utilizing the **Expression Builder**.

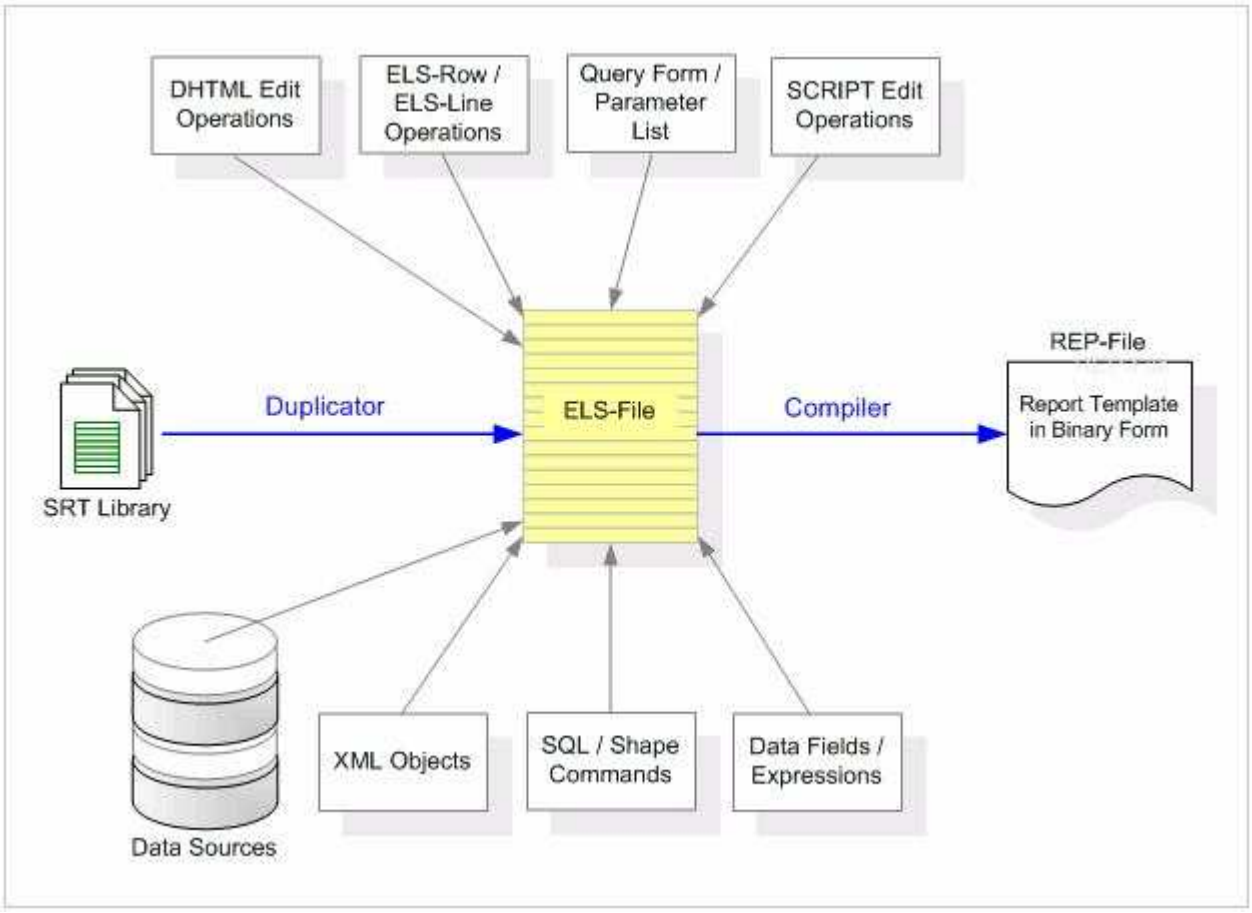

Figure 3.1. Showing the architecture of the report design and build processes

Along with data definition, data presentation processes are also involved in design of a report. In particular, given that ELS-Script is based on *Open Document Structure* technology with DHTML being the current target document format, the document typeset language becomes DHTML, including Java, VB and ASP scripting. Note that, given the fact that the SCRIPT language integrates relentlessly with ASP.NET, this inherent *ODS* technology permits the natural extension of ELS-Script to .NET platform.

Unlike other report tools in the current market, ELS-Script is designed around the concept of *Open Document Structure*, which briefly speaking exposes the internal report template format, making report development platform

very similar to scripting languages used in HTML documents. The *ODS* technology behind ELS-Script is in no way restricted to HTML or DHTML formats, and may be extended to any tag-oriented document format based on markup languages, such as *XML*, *MathML*, *VML* or *SVG*, to name a few.

HTML edit operations may be performed via the **Design** view of the *Report Designer* application, which is a multiple-pane sophisticated HTML editor comprising of operations to manipulate such features as text font, color, background color, border color and size, HTML table, other HTML elements along with attributes and styles, text alignment, image, paragraph style and indentation, as well as, other HTML features.

To define data tabulation, *ELS-Row* and *ELS-Line* operations may be used via the **Insert ELS-Row**, **Insert ELS-Shape** and **Insert ELS-Line** windows. Such tabulation constructs are more efficient than regular *HTML Tables*, and just like *HTML Tables*, they may be resized or manipulated via the **Design** or **Source** view after they are inserted into the report script. Using **Data Fields** and **Expression Builder** windows, the user may then insert data fields or expression about data fields into the cells of these tabulation constructs.

The **Query Form** designer aids the user in defining query form controls provided the **Report Generator** window is selected to be called by the host application. Otherwise communication from the host application to the report engine is assumed via the parameters defined in the **Parameter List** window of a report script.

Supplementing all these tools, almost about anything may be accomplished directly via the edit operations in the SCRIPT *Editor* window or the **Source** view.

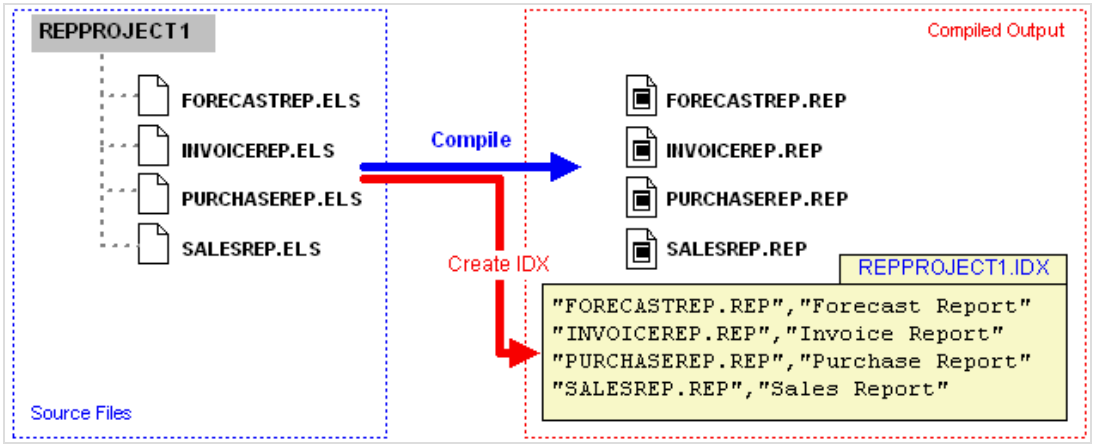

Figure 3.2. Showing an example of report project Build process

Report *Build* process parses and compiles all the report script *ELS*-files into corresponding binary *REP*-files. Moreover, it retrieves all the *REPORT_TITLE* values from the report settings section of the report scripts and creates an index *IDX*-file. In this way at run-time the report engine needs not re-parse the report script, but rather use the compiled object form contained in the corresponding binary *REP*-file. Observe that the format of the *IDX*-file is simply a comma delimited text file format, and comprises of two columns, the filename of the *REP*-file and the *REPORT_TITLE* of the report, see the example illustrated in Figure 3.2. The *IDX*-file is essentially used in the **Report Generator** window to define the content of the report list-box, in this way resembling a catalogue of reports in a project.

The *Build* process assumes that all report scripts for a project are located in the **REPORT** subfolder of the project folder, and when the process is run, the corresponding binary *REP*-files will be put into the **BIN** subfolder of the project folder along with the *IDX*-file. All images or related include files will be duplicated inside the **BIN** subfolder, as illustrated in Figure 3.3 below. Note that if the **BIN** subfolder does not yet exist in a project folder, the *Build* process will create the subfolder along with the **IMAGE** and **INCLUDE** subfolders of the **BIN** subfolder. The major advantage of such file management configuration is to simplify the deployment of the resulting *REP*-files integrating them with the host application. Apparently, what the developer needs to do is simply copy the whole content of the **BIN** subfolder to the selected target location of the host application. And therefore, unlike other web design applications, the *Report Designer* does not inherit any tedious file dependency problems.

To summarize this section, we list several other tools which are also important in the creation and design of reports. The **Format/Conversion Wizard** window is an indispensable tool for formatting variable values into

string expressions, as well as converting string values to date-time and numeric expressions. The **Connection/ Datasource List** window may be used to display all connection and data source objects in the current report script and navigate by a click to the location of the object's definition in the script to edit. The **Special Symbols** window contains a list of commonly used symbol characters. The **Insert Table** and **Insert Image** dialogs may be used respectively to insert *HTML Tables* and images in the report script. The **SQL Dictionary** contains detailed information about SQL elements along with proper syntax for almost all kinds of backend database types including MS-Access, MS-SQL Server, DB2, Sybase, FoxPro and MySQL. The **Find In Files** is a profound text search tool spanning over multiple files under specified path. Finally, the *IntelliSense* in the **Source** view's editor, as well as the **Expression Builder**'s editor greatly simplify the task of SCRIPT coding.

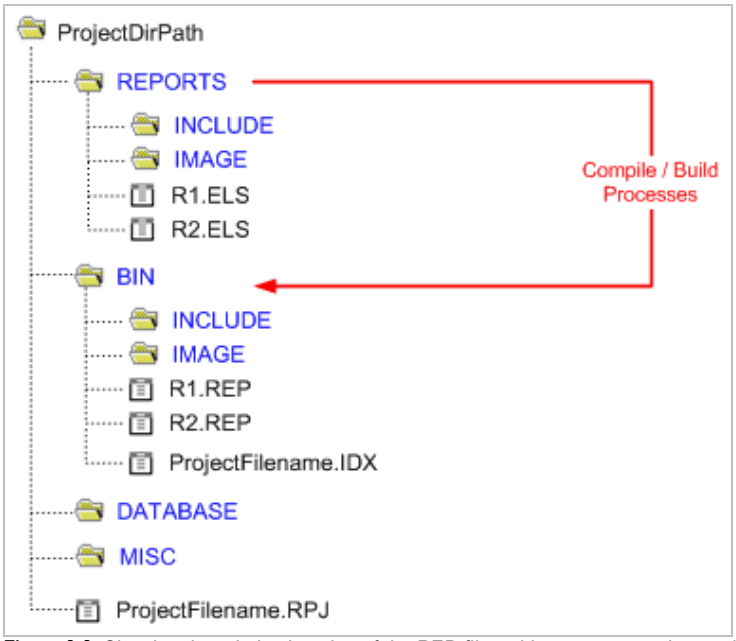

Figure 3.3. Showing the relative location of the REP-files with respect to project path

Report Generation

The report engine is optimized for both speed and memory consumption. Therefore, in order not to exhaust valuable system memory resources, the report output is generated into a simple indexed database, with each record being a page of the output. These database files have extension STG and essentially are temporary storage files that are deleted whenever a new generation is initiated for the same instance of the report engine, or whenever the engine instance is destroyed. Moreover, each generation instance will create a separate and unique *STG*-file. This later feature makes the report engine suitable for centralized multi-user environment, especially when deployed as a server-based web application or web service.

The SCRIPT report engines come with two modes of report generation,

- *ELS_FAST* mode, which essentially is the *paged* mode that we just described
- *ELS_CONTINUOUS* mode, which is the *continuous* mode

In the *continuous* generation mode, the report output is generated as a single continuous HTML or DHTML document, strictly preserving the format of all the HTML elements as they were in the source document. Because of this nature, unlike the *paged* generation mode, the report output in the *continuous* mode case is generated into a HTML file with the standard HTM file extension. Moreover, each such output *HTM*-file will have a unique filename depending on the generation instance, as well as, the report engine instance. Therefore, some advantages of the *continuous* generation mode are:

- Full compliance and preserving of the HTML or DHTML elements in HTML document, especially the width of *HTML Tables*, as well as, the *DIV* and *SPAN* sections
- Output suitability for web reporting as a single continuous web page
- Suitability for mail or document merge

In contrast, some of the advantages of *paged* mode are as follows:

- Suitability for printer based paged format
- Splitting of large report output into small page sizes suitable for Internet based low-bandwidth communication environment
- Minimize consumption of valuable system memory resources
- Maximize report generation speed
- Preserving the ODS structure to the extend of almost full compliance

The *ELS_FAST paged* generation mode may be deployed in a very effective way in a server-based ASP application. Essentially, one may define a web page where the generation is triggered, and the report output pages may be viewed in another web page consisting of two HTML frames, the upper frame may serve as a toolbar with navigation buttons, while the bottom frame will display the pages of the report output. Now when the user clicks on the next page button, a request is send from the client-browser to the ASP application, which asks for the next output page via the *GetPageContent(nPage)* API function of the report engine. This will get the content of *nPage* page of the output, which may be wrapped in the *Response* object and sent back to the client browser, displaying it in the lower HTML frame. The diagram in Figure 3.4 illustrates these mechanisms.

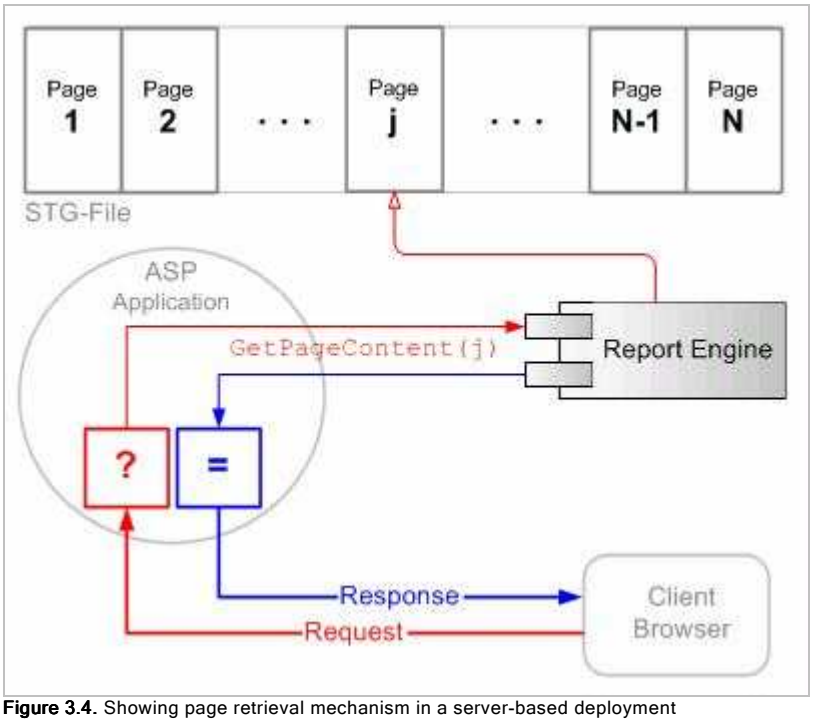

Figure 3.4. Showing page retrieval mechanism in a server-based deployment

Note that, since the report output pages are HTML documents, the client-browser will not need any plug-in or COM component, and therefore this deployment of SCRIPT engine is truly a zero-client-install web reporting solution.

Utilizing SCRIPT Engine

In this section we will describe various ways that one can utilize the SCRIPT engines. Recall that there are two types of SCRIPT engines, namely the **GEV**-engine and the **E**-engine. In addition to all the features of **E**-engine, the **GEV**-engine contains built-in **Report Generator** and **Report Viewer** windows, included in the component with the intension to simplify the task of integration of the report engine with the host application.

There are three ways to integrate the **GEV**-engine with a host application depending on end-user requirements, the application platform and the deployment environment.

(a) Utilizing the **Report Viewer** via the **Report Generator** window
(b) Utilizing the **Report Viewer** without calling the **Report Generator**
(c) Utilizing another target browser to view reports

The main goal of a developer using method (a) would be to minimize the implementation efforts needed in the integration of the report engine with the host application. In this case, all one has to do is to set some properties of **Report Viewer** and **Report Generator** windows via the SCRIPT engine APIs, and then call the **Report Generator** window. All other report generation tasks will be handled via the **Report Generator** with no extra effort.

Recall that the **Report Generator** window consists of the *report list* grid and the *query form* section. This *query form* section of the **Report Generator** window will display the controls defined in the *ELS-QPARAMS* section of the report that is selected in the *report list* grid. Therefore, it is not difficult to see that by including query parameter controls in the *ELS_QPARAMS* section of a report, the data selection information of a report is ingeniously put in the backend instead of the front-end. In traditional report tools, often you need to define data selection information in the host application, so that for each report type you will need to perform some implementation in the host. A drawback of this is that whenever some structural changes are needed in the report template, the host application will need a recompilation. None of the report tools that exist on the market have an option to include data selection control in the report template itself. In this sense, ELS-Script is very unique to allow such a configuration, so that new report templates may still be added to a host system long after the last compile and deployment of the host application.

To summarize, we note that the main advantages of method (a) are as follows:

- Simplifies the integration of report engine with host application.
- Makes report parameters independent from the host application by putting the data selection controls inside the report template itself.
- Saves the developer a fair amount of report specific coding in the host application.

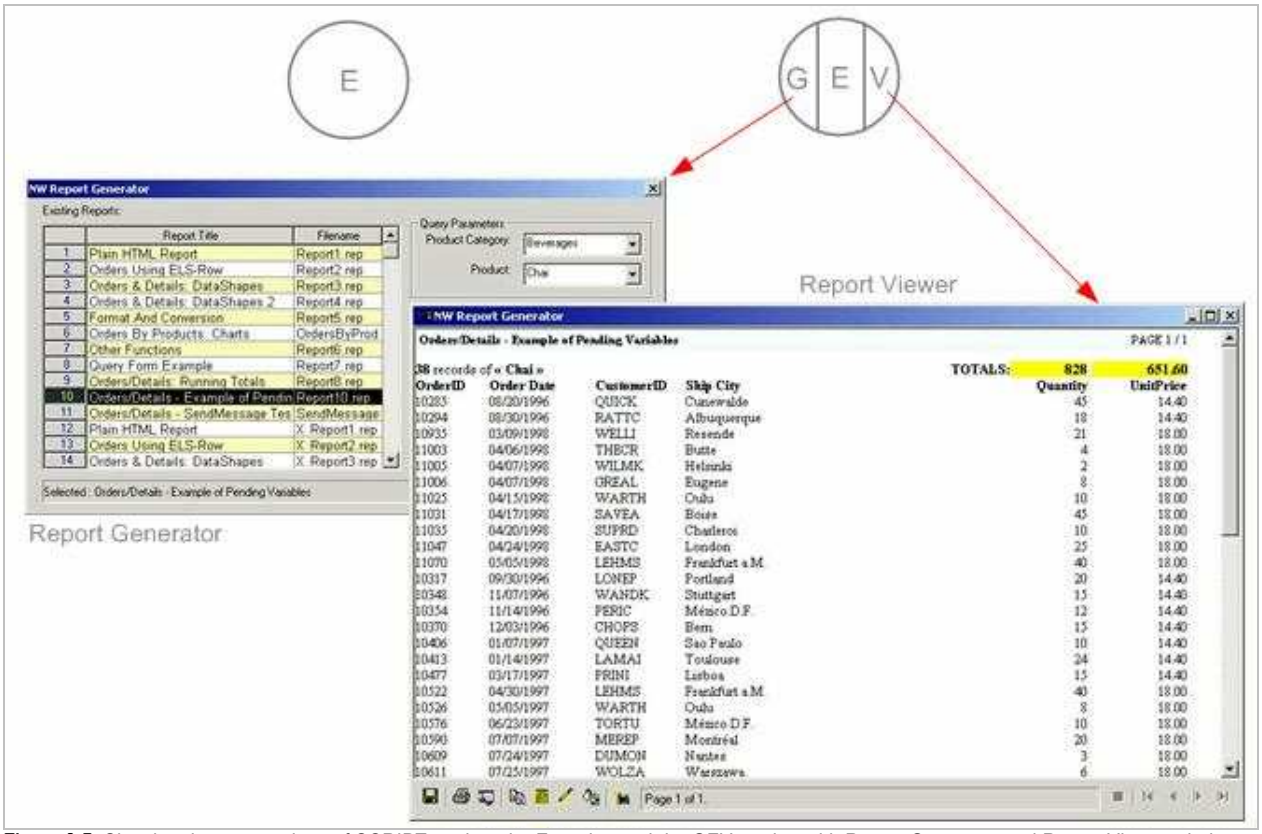

Figure 3.5. Showing the two versions of SCRIPT engine, the E-engine and the GEV-engine with Report Generator and Report Viewer windows

We next move on to the alternative method described in (b). In this case, the **Report Viewer** window is used without utilizing the **Report Generator**. This method may be necessary especially when it is required to develop a

non-standard data selection interface in the host application, from where the report generation is triggered. Observe that the existence or the possibility of method (c) in the integration of the report engine, in which case any target browser component may be used as a report output viewer. For example, one may use any HTML viewer component in the host application and target all report output into this viewer. Nevertheless, for most windows applications method (b) is recommended, since the developer will avoid a fair amount of coding in the host application by simply retaining the **Report Viewer** window. We should also emphasize that the **Report Viewer** is equipped with sophisticated built-in features, which makes it worth to be seriously considered whenever possible.
The **Report Viewer** window essentially consists of a view area and two toolbars at the bottom of the window. The view area is essentially a HTML viewer and is used to display the pages of the report output, while the two toolbars consist of the following buttons:

Save As	this will save the entire report output as HTML document to a specified location,
Print	this will call the printer dialog, so that the user may print the report output,
Print Preview	this will display the page in print preview window,
Copy	this will copy the selection to the clipboard,
Select All	this will select the content of the page,
High-light	this will high-light the selection,
Color Selector	this will display the high-light color dialog,
Find	this will call the **Find** window for text search over the entire report output,

for the left side toolbar, and

Stop	this will stop any pending report generation process,
First Page	this will display the first page of the report output in the viewer,
Previous Page	this will display the previous page of the report output in the viewer,
Next Page	this will display the next page of the report output in the viewer,
Last Page	this will display the last page of the report output in the viewer,

for the right side toolbar. In between these two toolbars, the **Report Viewer** displays the report generation status information or the current page number information.

One of the major advantages of the *ODS* compliance of ELS-Script is the fact that the report output pages are essentially HTML documents, and therefore may be viewed by any web browser. This fact by itself makes the method in (c) very possible. This also proves that ELS-Script software system is the best and most natural solution to web reporting.

Finally, we note that in method (c) we have utilized neither the **Report Generator** nor the **Report Viewer**, and therefore in such a case it is more suitable to use the **E**-engine rather than the **GEV**-engine. This is because the **E**-engine contains only the report engine with no GUI elements exposed, and for that matter is much smaller in size compared to the **GEV**-engine.

APIs of SCRIPT Engine

The SCRIPT engines are interfaced via a collection of API functions and objects. Since there are two types of SCRIPT engine, namely **E**-engine and **GEV**-engine, we describe first all the APIs that are common to both types of SCRIPT engines, and later on we continue this description with the rest of the APIs specialized to the **GEV**-engine types.

The main API object, common to both types of engines, is the *ELSReportEngine* class, which essentially handles all engine operations such as data access connection, report generation and storage. We outline next the properties, methods and events of *ELSReportEngine* class:

Connection	Property of type *Unknown*, this property defines the default connection for the report engine, note that the default connection will be used for the report generation if no further specifications occur in the *ELS*-file.
Compression	Property of type *Integer*, this property defines the HTTP 1.1 compression level to be used in the report engine. It can assume one of the following enumerated values:

```
ELS_NOCOMPRESSION = 0,
ELS_ONLYMPCOMPRESSION = 1,
ELS_FULLCOMPRESSION = 2,
```

By default the compression is set to `ELS_NOCOMPRESSION`.

TargetBrowser Property of type *Unknown*, this property defines the target browser in which the report output will be generated.

SetBaseURLPath(strURLPath) Subroutine, this function sets an explicit URL used to resolve links and references to external sources, such as images or hyperlinks used in the report. This URL path is necessary especially when deploying the engine in a server-based web application.

SetStoragePath(sPath) Subroutine, this function sets the storage directory used for temporary files created during report generation. By default, if this path is not defined the TEMP subdirectory of the executable directory will be the storage directory.

Initialize(bLogIsOn) Function with return type *Long*, this function initializes the report engine. The optional *Boolean* argument *bLogIsOn* specifies whether a report generation log file is created per report type or not. By default *bLogIsOn* is *FALSE*, so that if not specified then no log file will be created during generation of reports.

UnInitialize() Subroutine, this function un-initializes the report engine.

GenerateReport(objRep, CmdShow)
Function with return type *Long*, this function triggers the report generation process for the report specified by the *ELSReport* object *objRep*. The argument *objRep* is of type *Unknown*, the *CmdShow* is an optional *ECmdShow* type argument with default *ELS_SHOW*. Note that variables of *ECmdShow* type have two possible values, *ELS_SHOW* to show the report output, and *ELS_HIDE* to hide the report output generating it in the internal storage. In this later case, the content of the output may be retrieved via the *GetPageContent*, *GetOutput*, *SendOutputHTTP* and *SaveOutput* functions. This function, if successful will return a positive long value representing the actual number of generated pages. Otherwise, it will return 0 or a negative error value if unsuccessful.

Stop() Subroutine, this function will stop an active report generation process.

GetPageContent(nPage) Function with return type *String*, this function will return the content of the report output page specified by the *nPage* argument. In particular, it assumes that the report is already generated and the storage active. Note that the argument *nPage* is a variable of type *Long*.

GetPageSetupInfo(dLeftMargin, dRightMargin, dTopMargin, dBottomMargin, dPageSource, dPaperSize, dOrientation, dDefFontSize)
Function with return type *Long*, this function is used to get margin, page source, paper size, orientation and default font size information via the arguments, to be used by the host application (for example to set the page setup or printer setup for the currently generated report output). All arguments are variables of type *Double*. The *GetPageSetupInfo* function is an alternative to the *ReportPageInfo* event, and may be used for host platforms that cannot expose events.

GetOutput() Function with return type *String*, this function will return the whole content of the report output as a single HTML document. In particular, it assumes that the report is already generated and the storage active.

SaveOutput(sFilename) Subroutine, this function will save the whole content of the report output as single HTML document in the file specified via the *sFilename* argument.

Note that the argument *sFilename* is a variable of string type.

SendOutputHTTP(objResponse, [nStartPage], [nEndPage], [lReserved])
Function with return type *Long*, this function will return the number of pages of the output between the *nStartPage* and *nEndPage* values. The HTTP 1.1 compressed content of these pages will be returned via the *objResponse* variable of *Unknown* data type. The other arguments are all optional and have *Long* data types. If *nStartPage* and *nEndPage* are not specified, by default the whole report output will be returned in HTTP 1.1 compressed form.

SetBaseMPrintSupportURL(sURLPath)
Subroutine, this function specifies an explicit URL that will be used to resolve references to custom print templates.

BeginReport() — Event, this event is triggered just when the report generation is initiated but before the generation starts.

BeginPage(nPage) — Event, this event is triggered whenever a new page is about to start in the report generation process. The page number of the new page is passed to the host via the *nPage* argument of type *Long*.

EndPage(nPage) — Event, this event is triggered whenever a page is about to end in the report generation process. The page number of the ending page is passed to the host via the *nPage* argument of type *Long*.

EndReport() — Event, this event is triggered just after the report generation is completed.

Cancelled() — Event, this event is triggered when the report generation is interrupted by the host action.

ReportPageInfo(dLeftMargin, dRightMargin, dTopMargin, dBottomMargin, dPageSource, dPaperSize, dOrientation, dDefFontSize)
Event, this event is triggered just after a successful *BeginReport* event and before the firs *BeginPage* event. It returns margin, page source, paper size, orientation and default font size information via the arguments, to be used by the host application (for example to set the page setup or printer setup for the currently generated report output). All arguments are variables of type *Double*.

SendRGMessage(nMsg) — Event, this event is triggered whenever a user-defined message is passed from the report engine to the host as a result of *SendMessage* call in the report script. The argument *nMsg* is a variable of type *Long*.

Terminated(sCause, sSourceFilename, nLineNumber)
Event, this event is triggered whenever the report generation is terminated as a result of a run-time error, in which case the error cause, the *ELS*-file's filename and the line number are passed to the host respectively via the *sCause*, *sSourceFilename* and *nLineNumber* arguments.

The next common API object that we will describe is the *ELSParams* class. This class essentially handles the collection of all the parameter options defined in the `PARAM_OPTIONS` list of a report script. It has the following properties:

Count — Property of type *Long*, this property is read-only and returns the number of parameters defined in the selected *REP*-file.

Name(nIndex) — Property of type *String*, this property is read-only and returns the name of the parameter specified via the *nIndex* argument of type *Long*.

Type(nIndex, nType) Property of type *Long*, this property is read-only and returns the array size of the parameter specified via the *nIndex* argument or 0. It also passes the data type of the parameter as an integer value via the *nType* argument.

Possible *nType* values returned by the *Type* property are as follows:

nType Value	Data Type	Description
0	*Void*	Void data type
1	*Bit*	Integer data with either 0 or 1 value
2	*Int*	Integer data (Long)
3	*Smallint*	Integer data (Short)
4	*Tinyint*	Integer data (Byte)
5	*Numeric*	Numeric data
6	*Money*	Monetary data
7	*Smallmoney*	Monetary data
8	*Float*	Floating precision numeric data
9	*Real*	Floating precision numeric data
10	*Datetime*	Date-time data
11	*SmallDatetime*	Date-time data
12	*Timestamp*	A database-wide unique number (same as SQL Server)
13	*UniqueID*	A globally unique identifier (same as SQL Server)
14	*Char*	Fixed-length non-Unicode string data
15	*Varchar*	Variable-length non-Unicode string data
16	*NChar*	Fixed-length Unicode string data
17	*NVarchar*	Variable-length Unicode string data
18	*Binary*	Fixed-length binary data
19	*VarBinary*	Variable-length binary data
20	*Datasource*	ELS *DATASOURCE* object type
21	*Connection*	ELS *CONNECTION* object type
22	*QueryOption*	ELS Query Option
23	*Variant*	Variant data type
24	*Null*	NULL
25	*Unknown*	*IUnknown* object type
26	*Dispatch*	*IDispatch* object type
27	*Unsupported*	Unsupported data type

Table 3.1. *Showing all possible data type values*

As we will see next in the description of the *ELSReport* class, that parameter information may be passed from host application to the report engine via the *Param* property of the *ELSReport* class. Moreover, information about the parameter options may be retrieved via the *GetParams()* function. In this respect, there is a remarkable and powerful feature of ELS-Script that makes the API control over the report template extremely flexible. Namely, almost any content feature in the report template may be controlled and evaluated from the host application via the *Param* property.

The *ELSReport* class has the following properties and methods:

ReportFilename Property of type *String*, this property defines the path and the filename of the binary *REP*-file.

Param(sParam) Property of type *Variant*, this property defines the value of user-defined `PARAM_OPTIONS` variable specified via the *sParam* string argument.

GetParams() Function with return type *ELSParams* object, this function is used to get the list of all the parameters together with equivalent data type information from the `PARAM_OPTIONS` section of the *REP*-file. So that at run-time the host application may use this parameter information to make further decisions.

GetQOHeight() Function with return type *Long*, this function returns the height of the minimum rectangular region that will contain all the query option controls defined in the `QUERY_OPTIONS` section of the *REP*-file. This function is useful only when used with the **Report Generator** window, which therefore applies only to **GEV**-engines.

GetQOWidth() Function with return type *Long*, this function returns the width of the minimum rectangular region that will contain all the query option controls defined in the `QUERY_OPTIONS` section of the *REP*-file. This function is useful only when used with

the **Report Generator** window, which therefore applies only to **GEV**-engines.

As we have seen in Chapter 2, that in version 3.0 of ELS-Script, we have extended the report parameter options to include *Prompt*, *ValidValues* and *DefaultValues* important properties. To be able to support such properties we have added new forms of *ELSParams* and *ELSReport* classes to the collection of the APIs, namely the *IELSParams2* and *IELSReport2*. In addition to these classes, we have also implemented and exposed new classes *ELSNamedCollection* and *ELSParamOption*.

The new class *IELSParams2* represents the collection of all the parameters used in the selected report, and has the following members:

Count	Property of type *Long*, this property retrieves the number of parameters use in the report script.
Item(nIndex)	Property that returns an object of type *ELSParamOption*, this property retrieves the *nIndex*-th parameter option used in the report script. The *nIndex* argument is of type *Long*.
Name(nIndex)	Property of type *String*, this property is retained for backwards compatibility with reports compiled with old versions. It retrieves the name of the *nIndex*-th parameter.
Type(nIndex, nType)	Property of type *Long*, this property is retained for backwards compatibility with reports compiled with old versions. It retrieves the type information of the *nIndex*-th parameter (see the definition of the old class *ELSParams*).

The *Item* property of the *IELSParams2* collection returns an object of type *ELSParamOption*. This later class represents an extended parameter option used in the report, and has the following members:

AllowBlank	Property of type *Long*, this property retrieves the `AllowBlank` value of the parameter from the report.
ArraySize	Property of type *Long*, this property retrieves the array size of the parameter used in the report.
DataType	Property of type *Integer*, this property retrieves the data type of the parameter from the report. Possible values are outlined in Table 3.1.
LabelAlign	Property of type *Integer*, this property retrieves the label alignment for the parameter in the report.
Length([nIndex])	Property of type *Long*, this property retrieves the data length of the parameter from the report.
MultiValue	Property of type *Long*, this property retrieves the `MultiValue` value of the parameter from the report.
Name	Property of type *String*, this property retrieves the name of the parameter from the report.
Nullable	Property of type *Long*, this property retrieves the `Nullable` value of the parameter from the report.
Precision	Property of type *Long*, this property retrieves the precision of numeric value of the parameter from the report.
Prompt	Property of type *String*, this property retrieves the label text of the parameter from the report.
Scale	Property of type *Long*, this property retrieves the scale of the numeric value of the

parameter from the report.

SourceType — Property of type *Integer*, this property retrieves the source type for the `ValidValues` and `DefaultValues` properties of the parameter. Possible values are: 0 for an explicit list, or 1 for a data source.

ValidValues — Property of type *Variant*, this property retrieves the valid values for a parameter from the report, and returns as a *ELSNamedCollection*.

DefaultValues — Property of type *Variant*, this property retrieves the default values for a parameter from the report, and returns as an *ELSNamedCollection*.

Value([nIndex]) — Property of type *Long*, this property defines or retrieves (i.e. *Write* and *Read*) the value for the parameter of the report. The *nIndex* is optional and applies only when the parameter has multi-values (i.e. *MultiValue* is non-zero).

The *ELSNamedCollection* class is a collection of *ELSNamedItem* objects, and has the following members:

Count — Property of type *Long*, this property defines the number of items in the collection.

Item(nIndex) — Property that returns an object of type *ELSNamedItem*, this property retrieves the *nIndex*-th item in the collection.

In turn, the *ELSNamedItem* class has the following members:

Name — Property of type *String*, this property defines the name of item.

Value — Property of type *Variant*, this property defines the value of item.

The *IELSReport2* class is the extended version of the *ELSReport* class, and has the following members:

Param(sParam) — Property of type *Variant*, this property defines the value of the parameter in the report specified by *sParam*. This parameter is retained for backwards compatibility.

Parameter(vParamIndex, [nValueIndex]) — Property of type *Variant*, this property defines the value of the parameter in the report specified variant *vParamIndex* and the optional *Long nValueIndex* arguments. The first argument may be either the name of the parameter or the 0-based index of the parameter. The second argument applies only when the parameter is multi-valued.

ReportFilename — Property of type *String*, this property defines the path and filename of the binary *REP*-file of the report.

GetParams() — Function that returns object of type *ELSParams*, this function is retained for backwards compatibility (use *GetParams2* for extended parameters).

GetParams2(oEngine) — Function that returns object of type *IELSParams2*, this function first generates the *ELS_QPARAMS* section of the report and then returns the list of all the parameters used in the report.

The next API object that we will describe is the *ELSReportViewer* class, which applies only to **GEV**-engine type. This class has the following properties and methods:

Left — Property of type *Long*, this property defines the left coordinate of the **Report Viewer** window in pixels.

Top — Property of type *Long*, this property defines the top coordinate of the **Report Viewer** window in pixels.

Height — Property of type *Long*, this property defines the height of the **Report Viewer** window in pixels

Width — Property of type *Long*, this property defines the width of the **Report Viewer** window in pixels.

Title — Property of type *String*, this property defines the text of the title-bar of the **Report Viewer** window. By default if this property is never specified the value will be "Report Viewer".

WindowMode — Property of type *Integer*, this property defines a combination for the window mode used in displaying the **Report Viewer** window. Possible combinations are defined by the *ERepViewerMode* enumeration, which has the following possible bit values:

ELSRV_MODAL	to make window modal,
ELSRV_MAXIMIZED	to make window maximized,
ELSRV_NOMINBTN	to hide window's min button,
ELSRV_NOMAXBTN	to hide window's max button.

By the default *WindowMode* = 0, which means a modeless and normal window with min/max/close buttons.
For example, in Visual Basic setting *WindowMode* to
ELSRV_MODAL + ELSRV_NOMINBTN + ELSRV_NOMAXBTN
will display the **Report Viewer** as a modal window without the min and max buttons.

PopupMenu — Property of type *Long*, this property defines the visibility of the popup menu in the **Report Viewer** window, by default the menu will be visible whenever the end-user clicks the right mouse button.

ToolbarButtons — Property of type *Integer*, this property defines the visibility of the toolbar buttons in the **Report Viewer** window. Possible combinations are defined by the *ERepViewerToolBtns* enumeration, which has the following possible bit values:

ELSRV_NOSAVEBTN	to hide the **Save** button,
ELSRV_NOPRINTBTN	to hide the **Print** button,
ELSRV_NOPVIEWBTN	to hide the **Print Preview** button,
ELSRV_NOCOPYBTN	to hide the **Copy** button,
ELSRV_NOSELALLBTN	to hide the **Select All** button,
ELSRV_NOHLITEBTN	to hide the **High-light** button,
ELSRV_NOCOLORBTN	to hide the **Color** selector button,
ELSRV_NOFINDBTN	to hide the **Find** button,
ELSRV_NOFIRSTBTN	to hide the **First** page button,
ELSRV_NOPREVBTN	to hide the **Previous** page button,
ELSRV_NONEXTBTN	to hide the **Next** page button,
ELSRV_NOLASTBTN	to hide the **Last** page button.

By default *ToolbarButtons* = 0, which means that all buttons of the toolbar will be visible. Note that the **Run** and **Stop** buttons are not controlled by this property and must be always visible.

WebBrowser — Property of type *Unknown*, this property is read-only and is used to expose the internal web browser of the **Report Viewer** to the host application.

Show(bShow, bCenter) — Function with return type *Long*, this function is used to show or hide the **Report Viewer** window with the window mode defined via the *WindowMode* property. Both arguments are optional variables of type *Long*, and have by default *TRUE* values. The first argument controls the visibility of the window, while the second argument whether to center the window or not. Note that, the properties *Left*, *Top*, *Height* and *Width* apply only when the *bCenter* argument is *FALSE*.

The next API object that we will describe is the *ELSReportGenerator* class, which also applies only to **GEV**-engine type. This class has the following properties, methods and events:

Left	Property of type *Long*, this property defines the left coordinate of the **Report Generator** window in pixels.
Top	Property of type *Long*, this property defines the top coordinate of the **Report Generator** window in pixels.
Height	Property of type *Long*, this property defines the height of the **Report Generator** window in pixels.
Width	Property of type *Long*, this property defines the width of the **Report Generator** window in pixels.
Title	Property of type *String*, this property defines the text of the title-bar of the **Report Generator** window. By default if this property is not assigned the value will be "`Report Generator`".
RunButtonText	Property of type *String*, this property defines the text of the **Run** button in the **Report Generator** window. By default this property has the value "`Run`".
CancelButtonText	Property of type *String*, this property defines the text of the **Cancel** button in the **Report Generator** window. By default this property has the value "`Cancel`".
ReportLabelText	Property of type *String*, this property defines the text of the report list label in the **Report Generator** window. By default this property has the value "`Report:`".
ReportEngine	Property of type *ELSReportEngine* object, this property defines the report engine used by the **Report Generator** window.
ReportList	Property of type *ELSReportList* object, this property defines the report list used in the **Report Generator** window.
Show(bShow, bCenter)	Function with return type *Long*, this function is used to show or hide the **Report Generator** window. Both argument are optional variables of type *Long*, and have by default *TRUE* values. The first argument controls the visibility of the window, while the second argument whether to center the window or not. Note that properties *Left*, *Top*, *Height* and *Width* apply only when the *bCenter* argument is *FALSE*.
OnRun()	Event, this event is triggered when the user clicks the **Run** button in the **Report Generator** window, and just before the **Report Viewer** displays, and the report generation starts.
OnCancel()	Event, this event is triggered when the user clicks the **Cancel** button in the **Report Generator** window and just before the window hides (the *Close* button of the window will also trigger this same event).

The *ReportList* property is an instance of the *ELSReportList* class, which has the following properties and methods:

Visible	Property of type *Long*, this property defines the visibility of the report list in the **Report Generator** window. By default the report list is visible.
Height	Property of type *Long*, this property defines the height (in pixels) of the report list inside the **Report Generator** window.
Width	Property of type *Long*, this property defines the width (in pixels) of the report list inside the **Report Generator** window.
AlternateRowColor	Property of type *Long*, this property defines the background color of the alternate row of the report list grid. By default *AlternateRowColor* = *RGB*(255, 255, 180).

IDXPath	Property of type *String*, this property defines the path where the *IDX*-files are located. Note that if there are multiple *IDX*-files in a directory, then the content of all will be concatenated and displayed in the report list of the **Report Generator** window.
Font	Property of type *ELSListFont* object, this property defines the style, color and size of the font of the report list grid.
ColumnRatio(nCol1Width)	Subroutine to define the width ratio in percent of the two columns of the grid in the report list, namely the **Report Title** and **Filename**. The argument *nCol1Width* will define the percentage of the first column, for example setting *nCol1Width* to 60, will make the **Report Title** column width 60% of the grid width, while the second column, **Filename** will have width 40%. In particular, setting *nCol1Width* = 100, will hide the **Filename** column.
OnSelChange(sFilename)	Event, this event is triggered whenever the user changes the selection in the report list and just before the query option controls are updated on the right side of the **Report Generator** window. The argument *sFilename* will pass the filename of the report selected in the report list grid to the host application.

Finally, the *Font* property of the *ELSReportList* class is an object of type *ELSListFont* class, which has the following properties:

Style	Property of type *String*, this property defines the name of the font type. The default value is "`Arial`".
Size	Property of type *Long*, this property defines the size of the font in points. The default value is 10pt.
Color	Property of type *Long*, this property defines the color of the font in RGB. The default value is RGB(0, 0, 0), which is black.

GEV-engine Usage

In this section we illustrate the usage of the **GEV**-engine type of the report engine in the three methods described in the previous sections, namely methods (a), (b) and (c). All the samples in this section will be in Visual Basic.

Sample applications in other host languages, such as ASP and VB.NET, will be included in the later chapters of this book. For reasons of portability all of the samples and report scripts will use the Northwind.MDB MS-Access database. The user may easily change the backend database by modifying the connection string that is hard-coded in the sample's source code. For completeness, we have included full source code together with report scripts and a copy of the Northwind.MDB database, in the Samples subdirectory of the installation directory of the ELS-Script software package.

We will start with the scenario described by method (a). Recall that in this method, we want to use the **Report Viewer** window via the **Report Generator** dialog. The source code of the Visual Basic sample that hosts such a scenario is included in the Samples\VB\REPGEN_GEV subdirectory of the directory where the ELS-Script software is installed. In particular, we suggest that the reader opens the VB project file RepGenGEV.vbp in this subdirectory and follow the source code in parallel with the outline presented here.

After opening the VB project RepGenGEV.vbp, the first thing that we should observe is that the report engine ELSRepGenGEV 2.0 Library is included in the **References** dialog of the VB project. The application consists of an MDI form with menus and toolbar commands. The **Open Project** menu command basically utilizes the open file window's common dialog to specify the *IDXPath* parameter of the **Report Generator** window, which is called from inside the **GEV**-engine. Therefore, the main code is in the VB form frmMDI, which essentially handles this MDI interface with all required operations.

The code in the frmMDI form starts with the declaration of some global variables and objects. In particular, the

following objects are declared:

```
' Global variables
Public g sProjectFilePath As String     ' IDXPath
Public g_ConnString As String           ' connection string
Public g oConn As New ADODB.Connection  ' ADO connection object
Public WithEvents g_oRepGen As ELSReportGenerator   ' Report Generator window
Public WithEvents g_oRepEngine As ELSReportEngine   ' Report Engine instance
Public WithEvents g_oRepList As ELSReportList       ' Report List object
```

Note that there is no need for an *ELSReportViewer* object since the **Report Viewer** window is called internally via the **Report Generator** window. The VB code continues with declaration of form level variables followed by the *MDIForm_Load* subroutine, in which the toolbar and status-bar properties are defined followed by the initialization of the already declared global objects, and finally the connection string `g_ConnString` is set.

```
    ' Initialize global variables
    Set g oConn = Nothing
    Set g_oRepGen = Nothing
    Set g_oRepEngine = Nothing

    ' Define the connection string value
    ' (Please modify connection string value if you have a different data source path)
    g ConnString = "Provider=Microsoft.Jet.OLEDB.4.0;
                  Data Source=C:\ELSS\Samples\Data\Northwind.mdb;Persist Security Info=False"
```

The next subroutine that is called by the *OpenProject()* procedure is the *InitGenerator()*, which initializes the *ELSReportGenerator* object by setting all its properties, as illustrated in the code snippet below:

```
Public Sub InitGenerator()
    With g_oRepGen
        .Height = 340
        .Width = 644
        .Title = "NW Report Generator"
        .ReportLabelText = "Existing Reports:"
        .RunButtonText = "OK"
        .CancelButtonText = "Close"
        .ReportList.Height = 246
        .ReportList.Width = 360
        .ReportList.AlternateRowColor = RGB(230, 230, 255)
        .ReportList.ColumnRatio 70
        .ReportList.Font.Color = RGB(0, 0, 0)
    End With
End Sub
```

The *OpenProject()* subroutine is triggered via the **Open Project** menu item under the **File** menu, and has the following main code lines:

```
    Dim bAttachEngine

    bAttachEngine = (g_oRepGen Is Nothing)
    If Not g_oRepGen Is Nothing Then
        g_oRepGen.Show ELS_HIDE
    Else
        Set g_oRepGen = New ELSReportGenerator
        InitGenerator
    End If
    With commonDlgOpen
        .Filter = "Project Files (*.idx)|*.idx"
        .CancelError = True
        .DialogTitle = "Open project file:"
        .Flags = cdlOFNHideReadOnly Or cdlOFNOverwritePrompt
        .ShowOpen
        On Error GoTo 0
        g_oRepGen.ReportList.IDXPath = .FileName
```

```
        g_oRepGen.Show ELS_SHOW
        If bAttachEngine Then
            Set g_oRepEngine = g_oRepGen.ReportEngine
        End If
        Set g_oRepList = g_oRepGen.ReportList
        OpenConnection
    End With
```

As we have noted that this subroutine creates an *ELSReportGenerator* object and initializes this object's properties via the *InitGenerator()* subroutine. Then opening the *open file common* dialog it gets the *IDXPath* value from the user-specified path, and displays the **Report Generator** window. The subroutine proceeds by setting the `g_oRepEngine` and `g_oRepList` objects respectively to the *ELSReportEngine* instance and *ELSReportList* instance of the **Report Generator**. Finally the *OpenConnection()* subroutine is called, which essentially resets the ADO connection object `g_oConn`, and then opens this connection using the value of `g_ConnString` as connection string. If this connection is successful, then this connection object is passed to the internal report engine object of the `g_oRepGen` object. The following code snippet illustrates this process:

```
    ' Reset global connection variable
    ResetConnection
    On Error GoTo ErrConHandler

    ' Instantiate and open connection object
    Set g_oConn = New ADODB.Connection
    g_oConn.Open g_ConnString, "", "", adOpenUnspecified

    If g_oConn.State = adStateOpen Then
        ' Use client-side cursor
        g_oConn.CursorLocation = adUseClient
        ' Set the connection object of report engine to this open connection
        g_oRepGen.ReportEngine.Connection = g_oConn
    Else
        MsgBox "Connection failed."
    End If
```

With the outlined code lines, the host application will display the **Report Generator** window, which lists all the reports of the selected *IDX*-file in the report list. The end-user may at this point select a report from this list and hit the **OK** button (i.e. the *Run* button which has label text "`OK`") of the **Report Generator** window. Recall that when the report list selection is changed, events internal to the selected report's query form controls will be activated, in this way updating the query form section of the **Report Generator** window. Moreover, the end-user may set or enter values of these query form control prior to clicking the **OK** button, which will determine the query conditions for the report to be generated. When the report generation is triggered from the **Report Generator** window via the *OnRun()* event, the report engine's internal *IWebBrowser2* interface is passed to the *InitViewer* subroutine, which essentially becomes the instance of the *ELSReportViewer* class.

```
' Initialize the viewer whenever the user runs a new report via the Report Generator
Private Sub g_oRepGen_OnRun()
    ' Use the report engine's internal IWebBrowser2 interface as HTML viewer
    InitViewer g_oRepGen.ReportEngine.TargetBrowser
End Sub

' Define the initial values of the properties of Report Viewer window
Public Sub InitViewer(oViewer As ELSReportViewer)
    If Not oViewer Is Nothing Then
        With oViewer
            .Height = 700
            .Width = 770
'            .Top = 0
'            .Left = 0
            .Title = "NW Report Generator"
            .PopupMenu = True
'            .ToolbarButtons = ELSRV_NOPVIEWBTN + ELSRV_NOFINDBTN + ELSRV_NOPRINTBTN
'            .WindowMode = ELSRV_MAXIMIZED+ELSRV_NOMINBTN + ELSRV_NOMAXBTN
            .Show ELS_SHOW, ELS_CENTERED
        End With
    End If
```

```
End Sub
```

In the code listing above, the *g_oRepGen_OnRun()* subroutine calls the *InitViewer* subroutine, which initializes the **Report Viewer** window, setting all the properties and eventually displaying the window via the *Show* function of *ELSReportViewer* class. Note that the ELS_CENTERED value of the second argument will force the **Report Viewer** window to be centered ignoring the *Left* and *Top* property values.

Note that when the **OK** button is clicked, the report generation process is triggered internally via the **Report Generator** window, and therefore there is no need to call the *GenerateReport* function of the *ELSReportEngine* class to generate the selected report.

The final two important subroutines have the following code:

```
' Catch report engine error messages via Terminated event of report engine
Private Sub g_oRepEngine_Terminated(ByVal bstrCause As String,
                         ByVal bstrSourceFileName As String, ByVal nLineNumber As Long)
    MsgBox "Report generation Error:" & vbCrLf & vbCrLf & bstrCause
    stBarMDI.SimpleText = "Report Generation error."
End Sub

Private Sub MDIForm_Unload(Cancel As Integer)
    If Not Cancel Then
        ResetConnection
        ' both objects must be destroyed to clear STG-files
        Set g_oRepGen = Nothing
        Set g_oRepEngine = Nothing
    End If
End Sub
```

The form *Unload* procedure essentially forces the destruction of the `g_oRepGen` and `g_oRepEngine` objects, which is required to clear the *STG*-files for the last generated report. The *g_oRepEngine_Terminated* subroutine handles the *Terminated* event of the *ELSReportEngine* class. Recall that this event is triggered by the report engine whenever a run-time error occurs.

In order to test the RepGenGEV.vbp VB project, we will need a SCRIPT project with proper report scripts. The Samples\SCRIPT\TestProj1 subdirectory contains the TestProj1.RPJ SCRIPT project, which contains the following report scripts:

PlainHTML.ELS	example of report that makes use of *HTML Table* for tabular presentation,
BasicROW.ELS	example of report that makes use of *ELS-Row* for tabular presentation,
DataShape1.ELS	example of report that makes use of *ELS-Rows* and *Data Shapes*,
DataShape2.ELS	another example of a report using *ELS-Rows* and *Data Shapes*,
FormatTest.ELS	this report examines all possible *Format* combinations,
FunctionsTest.ELS	this report examines all the other SCRIPT functions,
OrdsByProdsWithChart.ELS	example of report that utilizes MS-Chart ActiveX component,
RunningTotals.ELS	example of report with running totals,
PendingVarTest.ELS	example of report that makes use of pending variables,
SendMessageTest.ELS	this report tests the *SendMessage* SCRIPT function,
NWInvoice.ELS	this report is the SCRIPT version of the standard Northwind invoice.

Opening this SCRIPT project and building all the reports will create the binary *REP*-files along with the *IDX*-file in the BIN subdirectory. Pointing the *open file common* dialog of the VB host application to this *IDX*-file, the user may be able to complete the testing of this sample application.

We consider now another Visual Basic sample application, which hosts the report engine following the scenario in method (b). Recall that in this alternative method, one is utilizing the **Report Viewer** window without using the **Report Generator**, in which case a custom parameter option form must be implemented in Visual Basic. The source code of the current sample application is included in the Samples\VB\REPVIEWER_GEV subdirectory of the root directory where the ELS-Script software is installed. The VB project RepViewerGEV.vbp contains the following main forms:

frmMDI	which is the main MDI form,
frmRepGen	which is the custom report generator VB form,
ConnectDlg	which is the data access connection definition dialog.

Since, in this case the standard built-in **Report Generator** of the *ELSRepGenGEV* engine will not be used, we need to plan the development of all the report scripts prior to the integration of the report engine with the host application. The SCRIPT project TestProj2.RPJ, under the C:\ELSS\Samples\SCRIPT\TestProj2 subdirectory, is used for this purpose, and contains to the following report script files:

PlainHTML.ELS	example of report that makes use of *HTML Table* for tabular presentation,
BasicROW.ELS	example of report that makes use of *ELS-Row* for tabular presentation,
DataShape1.ELS	example of report that makes use of *ELS-Rows* and *Data Shapes*,
DataShape2.ELS	another example of a report using *ELS-Rows* and *Data Shapes*,
OrdsByProdsWithChart.ELS	example of report that utilizes MS-Chart ActiveX component,
NWInvoice.ELS	this report is the SCRIPT version of the standard Northwind invoice.

Note that the OrdsByProdsWithChart.ELS and NWInvoice.ELS reports have been modified to interface with the host application. In particular, the query option variables in these two report script have been replaced by equivalent parameter option variables (check the *ELS_QPARAMS* sections of these report scripts for more details). The host application starts with the declaration of global objects and variables:

```
' Global objects and variables
Public g_sScriptDir As String
Public g_ConnString As String
Public g_bRepGeneration As Boolean
Public g_oConn As New ADODB.Connection
Public WithEvents g_oRepEngine As ELSReportEngine
Public g oRepViewer As ELSReportViewer
Public g oReport As ELSReport

' Form level variables
Public tbVisible As Boolean
Public stbarVisible As Boolean
```

Observe that in this application we did not declare any *ELSReportGenerator* objects, instead we declared a *ELSReportViewer* object. In the form load event, after the toolbar and status-bar construction, some of these global objects and variables are initialized and the path to the BIN subdirectory of the compiled reports is stored in the `g_sScriptDir` variable to be used later in the frmRepGen module.

```
    Set g_oConn = Nothing
    Set g oReport = Nothing
    Set g_oRepEngine = Nothing
    Set g_oRepViewer = Nothing

   ' Define path to the BIN subdirectory
    g_sScriptDir = "C:\ELSS\Samples\SCRIPT\TestProj2\BIN\"

    ' Define the connection string
    g_ConnString = "Provider=Microsoft.Jet.OLEDB.4.0;
                    Data Source=C:\ELSS\Samples\Data\Northwind.mdb;Persist Security Info=False"

    Set g_oRepEngine = New ELSReportEngine
    g oRepEngine.Initialize
```

The custom report list dialog is called via the **Open Report List** menu of the frmMDI VB form. This dialog is controlled by the frmRepGen VB form module, which starts with filling the list-box lstRepList with constant strings representing the reports, as shown in the form load event:

```
Private Sub Form Load()
    lstRepList.AddItem "Plain HTML Tabulation"
```

```
    lstRepList.AddItem "Basic ELS-Row Tabulation"
    lstRepList.AddItem "Orders/Details Data Shape 1"
    lstRepList.AddItem "Orders/Details Data Shape 2"
    lstRepList.AddItem "Orders By Products With Chart"
    lstRepList.AddItem "Nothwind Invoice"
End Sub
```

The frmRepGen form contains the following controls:

lstRepList	List-box control to display report names or description,
lblCombo1	Label control to show the parameter caption for the report selected in lstRepLits,
lblParamInfo	Label control to display the parameter structure of the report selected in lstRepList,
txtOrdID	Textbox control to enter *OrderID* value,
Combo1	Combo-box control to select chart type.

The txtOrdID and Combo1 controls are hidden by default, when the user selects the *Northwind Invoice* report in the lstRepList list-box, the txtOrdID textbox becomes visible and the caption of the lblCombo1 is updated to show "`Order ID :`". Similarly, when the *Order By Products With Chart* report is selected the Combo1 control becomes visible and the caption of the lblCombo1 becomes "`Chart Type :`". Note that the purpose of these mechanisms is to dynamically update the parameter options of the frmRepGen form depending on the report selection in the report list. The following code lines of the *lstRepList_Click()* subroutine illustrate these mechanisms:

```
    Dim sFilename As String

    lblCombo1.Visible = False
    Combo1.Visible = False
    txtOrdID.Visible = False
    ' Start with the path to BIN, then add report filename of select report
    sFilename = frmMDI.g_sScriptDir
    Select Case lstRepList.ListIndex
        Case 0: sFilename = sFilename + "PlainHTML.rep"
        Case 1: sFilename = sFilename + "BasicROW.rep"
        Case 2: sFilename = sFilename + "DataShape1.rep"
        Case 3: sFilename = sFilename + "DataShape2.rep"
        Case 4:
            sFilename = sFilename + "OrdsByProdsWithChart.rep"
            lblCombo1.Caption = "Chart Type :"
            lblCombo1.Visible = True
            Combo1.Visible = True
            ' Fill the combo with chart types
            Combo1.Clear
            Combo1.AddItem "2D Bar"
            Combo1.AddItem "3D Bar"
            Combo1.AddItem "2D Line"
            Combo1.AddItem "3D Line"
            Combo1.AddItem "2D Area"
            Combo1.AddItem "3D Area"
            Combo1.AddItem "2D Step"
            Combo1.AddItem "3D Step"
            Combo1.AddItem "2D Pie"
            Combo1 = "2D Bar"
        Case 5:
            sFilename = sFilename + "NWInvoice.rep"
            txtOrdID.Visible = True
            lblCombo1.Caption = "Order ID :"
            lblCombo1.Visible = True
    End Select

    Dim I, nCnt, lType As Integer
    Dim oParams As ELSParams

    ' Initialize the global ELSReport object
    Set frmMDI.g_oReport = Nothing
    Set frmMDI.g oReport = New ELSReport

    ' Set the filename, identify the report template
    frmMDI.g_oReport.ReportFileName = sFilename
    ' Get all parameter options from the report template
    Set oParams = frmMDI.g_oReport.GetParams
```

```
    Dim sItem As String
    Dim lArrSize As Long
    nCnt = oParams.Count
    ' For all parameter options get the name and data type
    For I = 0 To nCnt - 1
        lArrSize = oParams.Type(I, lType)
        sItem = sItem & oParams.Name(I) & " : " & DataTypeName(lType) & vbCr
        If lArrSize > 1 Then
            sItem = sItem & oParams.Name(I) & "(" & Mid$(Str(lArrSize), 2) & ") : "
                                    & DataTypeName(lType) & vbCr
        End If
    Next I
    lblParamInfo.Caption = sItem
```

Besides dynamically setting the appropriate parameters, the *lstRepList_Click()* subroutine also sets the *ELSReport* object and retrieves the parameter option information from the selected report template.
When the user selects a report from the lstRepList report list-box, sets the proper parameters, and then clicks on the **OK** button, the *cmdRun_Click()* subroutine gets activated. This subroutine, essentially checks if data access connection is established, if a connection is not open it calls the connectDlg form to connect. It then proceeds to check the proper parameter values and utilizing these values, the procedure finally calls the *GenerateReport* subroutine of the frmMDI module. This is illustrated in the following code lines:

```
Private Sub cmdRun Click()
    ' Call the connection dialog if no connection is open
    If frmMDI.g oConn.State <> adStateOpen Then
        connectDlg.Show vbModal
    End If

    ' Validate parameter entry and set g_oReport parameter to valid value
    If lstRepList.ListIndex = 4 Then
        frmMDI.g_oReport.Param("ChartType") = Combo1
    ElseIf lstRepList.ListIndex = 5 Then
        If txtOrdID = "" Then
            MsgBox "Please enter some value for Order ID!"
            txtOrdID.SetFocus
            Exit Sub
        End If
        frmMDI.g_oReport.Param("OrderID") = txtOrdID
    End If
    ' Call the main generation process
    frmMDI.GenerateReport
End Sub
```

The *GenerateReport* subroutine has the following code lines:

```
Public Sub GenerateReport()
    On Error Resume Next
    ' Initialize the report viewer object
    If Not g_oRepViewer Is Nothing Then
        Set g oRepViewer = Nothing
    End If
    Set g_oRepViewer = New ELSReportViewer
    ' Set the properties of the viewer object
    InitViewer g_oRepViewer
    ' Set the web-browser interface of the report engine to the viewer object
    g_oRepEngine.TargetBrowser = g_oRepViewer
    ' Call the GenerateReport API function to generate report
    g_oRepEngine.GenerateReport g_oReport, ELS_SHOW
End Sub
```

We complete the outline of the source code of the current host application by listing the event handlers for some of the events of the report engine.

```
Private Sub g_oRepEngine_BeginReport()
```

```
    stBarMDI.SimpleText = "Starting Report Generation"
    g_bRepGeneration = True
End Sub

Private Sub g_oRepEngine_EndReport()
    stBarMDI.SimpleText = "Report Generation completed"
    g_bRepGeneration = False
End Sub

Private Sub g oRepEngine Terminated(ByVal bstrCause As String, ByVal bstrSourceFileName As
String, ByVal nLineNumber As Long)
    MsgBox "Report generation Error:" & vbCrLf & vbCrLf & bstrCause
    stBarMDI.SimpleText = "Report Generation error."
    g_bRepGeneration = False
End Sub

Private Sub g_oRepEngine_Cancelled()
    stBarMDI.SimpleText = "Report Generation terminated by user."
    g_bRepGeneration = False
End Sub
```

The *BeginReport*, *EndReport* and *Cancelled* event handlers, in this case, just control the global variable `g_bRepGeneration` depending on the start, end or cancellation of the report generation process. The *Terminated* event handler essentially catches any run-time error that may occur during the report generation process.

E-engine Usage

In the next samples we will use a design that follows method (c). Recall that in this method the **Report Generator** and **Report Viewer** windows are not used, instead the report output must be viewed in an alternative web browser. This makes the *ELSRepGenE.DLL* (i.e. type **E**-engine) component a more suitable report engine than the overly sized *ELSRepGenGEV.DLL*. In fact, for web reporting the **E**-engine is the recommended engine, simply because of its smaller footprint and internal HTTP 1.1 compression. Therefore, in all the samples that follow, we will use the ELSRepGenE.DLL as our report engine.

The next sample application is a server-based ASP application, in which the report generation is performed entirely on the web server, and all that the client-side will need is the IE browser, or any browser that is compatible with ASP, Java and VB scripting.

To be able to run this server-based ASP application, use the *Internet Services Manager* to create a virtual directory. For example, select the **New** > **Virtual Directory** menu item of the **Action** menu. This will call the **Virtual Directory Creation Wizard** window, click the **Next** button to get to **Virtual Directory Alias** and enter the alias ELSS3ASPSample. Click **Next** to get to **Web Site Content Directory** and enter the path of the sample files, for example, C:\ELS\ELSS\Samples\ASP. Then click **Next** to get to **Access Permissions**, make sure the checkboxes **Read**, **Run scripts** and **Browse** are checked. Then click **Next** and then **Finish** to complete the setup.

The sample essentially consists of a main web page MainPage.htm, from where the end-user can jump to individual 2-framed web report viewer via the following hyperlinks:

Orders By OrderID Report
Orders By Date Range Report
Orders By OrderDate Report
All Invoices Report
Invoice
Sales Summaries with Charts
Report with Picture
Simple XML Report
Report with VML Chart

These hyperlinks call the ReportViewer.asp web page with respectively the following HTML anchor *HREF*-s:

```
href = "ReportViewer.asp?RepFile=/REPFolder2/OrdersByOrderID.rep&DBIndex=0"
href = "ReportViewer.asp?RepFile=/REPFolder1/OrdersByDateRange.rep
                    &ParamNames=BeginDate;EndDate&ParamValues=01/01/1995;06/01/1997"
href = "ReportViewer.asp?RepFile=/REPFolder1/OrdersByOrderDate.rep"
```

```
href = "ReportViewer.asp?RepFile=/REPFolder1/AllInvoices.rep"
href = "ReportViewer.asp?RepFile=/REPFolder1/Invoice.rep"
href = "ReportViewer.asp?RepFile=/REPFolder2/SalesSumChart.rep"
href = "ReportViewer.asp?RepFile=/REPFolder1/AbsPosPictReport.rep"
href = "ReportViewer.asp?RepFile=/REPFolder2/SimpleXMLReport.rep"
href = "ReportViewer.asp?RepFile=/REPFolder2/VMLChartReport.rep"
```

The actual SCRIPT source files of these compiled binary report *REP*-files may be found in the project folder for the WebSampleProject.RPJ report project, namely \Samples\SCRIPT\WebSampleProject subdirectory. Therefore, for further analysis about the content of these reports, you may open the WebSampleProject.RPJ report project via the *Report Designer* application and view the corresponding report scripts.

We start the outline of the source code for this sample with the *Session_OnStart* and *Session_OnEnd* session events, which are traditionally included in the global.asa file. In particular, in the *Session_OnStart* event, note that we create the *ELSReportEngine* and *ELSReport* objects and initialize the report engine and all global variables.

```
Sub Session_OnStart
    Session("PageCount") = 1            ' to handle the report output page count
    Session("CurrPage") = 1             ' to keep track of the current output page
    Session("InGeneration") = 0         ' in generation process or not
    Session("RepIndex") = 0             ' the index of the report file
    Session("DlgHeight") = 0            ' the parameter dialog's height
    Session("EngineConnected") = False          ' engine connected or not
    Session("ReportInitialized") = False        ' report initialized or not
    Session("RepName") = ""             ' to pass the report filename
    Session("ParamName") = ""           ' to pass the parameter name
    Session("DlgContent") = ""          ' to pass the parameter dialog's content
    Session("ShowParamsDialog") = 1     ' show parameter dialog or not
    Session("ParamNames") = ""          ' to pass the parameter names array
    Session("ParamValues") = ""         ' to pass the parameter values array
    Session("Conn") = Null              ' to handle the data connection object
    Session("ParamsObject") = Null      ' to handle the parameter objects used in report

    ' this assumes that the data is located outside the virtual directory location,
    ' therefore, we use the server's root path minus the string "\ASP"
    Dim sRootPath
    sRootPath = Server.MapPath(".")
    sRootPath = Left(sRootPath, Len(sRootPath)-4)

    ' constant data access connections for multiple databases
    Session("ConnStr0") = "Provider=Microsoft.Jet.OLEDB.4.0;" &
           "Persist Security Info=False;Data Source=" & sRootPath & "\Data\Northwind.mdb"
    Session("ConnStr1") = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
           "Persist Security Info=False;Data Source=" & sRootPath & "\Data\AnotherDB.mdb"
    ' ... add more ConnStr constants here as required ...

    ' puting the connection string information in global.asa will make it secure,
    ' and to select a defined particular connection, pass the DB index number from
    ' the main HTML page as a ReportViewer.asp parameter
    Session("ConnectString") = Session("ConnStr0")

    ' create the report engine and report object instances
    Set Session("ReportEngine") = server.CreateObject("ELSRepGenE.ELSReportEngine")
    Set Session("Report") = server.CreateObject("ELSRepGenE.ELSReport")

    If IsObject(Session("ReportEngine")) Then
           ' declare variables for file system objects
           Dim objFSO, objFile, objFileItem, objFolder, objFolderContent, TempDir

           ' initialize the report engine
           Session("ReportEngine").Initialize
           ' define the report engine's storage path
           TempDir = Server.MapPath(".") & "\Bin\Temp\"
           Session("ReportEngine").SetStoragePath TempDir

           ' create a file system object instance, this file system will be used to cleanup
           ' possible left-over STG storage files in the TempDir storage path
           Set objFSO = server.CreateObject("Scripting.FileSystemObject")
           Set objFolder = objFSO.GetFolder(TempDir)
           Set objFolderContent = objFolder.Files
```

```
            ' attempt to delete all the content of the storage path directory,
            ' note that the error mechanism is used to delete only the files
            ' for which access privilege is granted (i.e. not in use by others)
            For Each objFileItem In objFolderContent
                  On Error Resume Next
                  objFSO.DeleteFile TempDir & objFileItem.Name
                  If Err.Number <> 0 Then
                        Err.Clear
                  End If
            Next
    End If
End Sub
```

Observe in this code that all important steps of the routine are properly described via comment lines, and that essentially, the main functions of the routine are to declare session variables, initialize the report engine and report object instances, define the possible database connection strings, define the *STG* storage path, and finally perform some file cleaning operations.

In the *Session_OnEnd* event the connection object is destroyed, the report engine is un-initialized, and then the *ELSReportEngine* and *ELSReport* objects are destroyed, this is shown in the code below:

```
Sub Session_OnEnd
    If IsObject(Session("Conn")) Then
            If Session("Conn").State = adStateOpen Then
                  Session("Conn").Close
            End If
            Set Session("Conn") = Nothing
    End If
    If IsObject(Session("ReportEngine")) Then
            Session("ReportEngine").UnInitialize
    End If
    Set Session("ReportEngine") = Nothing
    Set Session("Report") = Nothing
End Sub
```

The call to the ReportViewer.asp starts with the following ASP code:

```
On Error Resume Next
Dim oOldParams

' check if ELSReportEngine and ELSReport object instances are created
If IsObject(Session("ReportEngine")) And IsObject(Session("Report")) Then
    If Request.QueryString("Action") = "re-run" Then
            Set oOldParams = Session("Report").GetParams()
            If Err.number <> 0 Then
                  Response.Write "Parameter access error: " & Err.Description
                  Err.Clear
            End If
    Else
            Set Session("Report") = Nothing
            Set Session("Report") = server.CreateObject("ELSRepGenE.ELSReport")
    End If

    ' if DBIndex is not specified then use the default database
    If Request.QueryString("DBIndex") = "" Then
            Session("ConnectString") = Session("ConnStr0")
    Else
            Session("ConnectString") = Session("ConnStr"+Request.QueryString("DBIndex"))
    End If

    sConnStr = Session("ConnectString")
    ' call the ConnectEngine function passing the connection string to the report engine
    Session("EngineConnected") = ConnectEngine(sConnStr)
End if

If Session("EngineConnected") Then
    ' define the path to the Phookctl.CAB file which contains the Printer class
    sBaseAddr = "http://" & Request.ServerVariables("SERVER_NAME") & _
```

```
                                                Request.ServerVariables("PATH_INFO")
sCabAddr = Left(sBaseAddr, InStrRev(sBaseAddr, "/", -1, 1)) & _
                                                "BIN/Phookctl.CAB#Version=1,0,0,4"

' attempt loading the report
bLoadSuccess = LoadReport()
If bLoadSuccess Then
        ' set the session variable RepName to the RepFile parameter passed to this page
        Session("RepName") = Request.QueryString("RepFile")
        ' use the GetParams2() API function to retrieve report parameters from the REP-file
        Set Session("ParamsObject") = Session("Report").GetParams2(Session("ReportEngine"))
        If Err.number <> 0 Then
                Response.Write "Generation Error: " & Err.Description
                Err.Clear
                Session.Abandon
        End If
        If Request.QueryString("Action") <> "re-run" Then
                Session("ParamNames") = Request.QueryString("ParamNames")
                Session("ParamValues") = Request.QueryString("ParamValues")
        End If

        ' split the parameters specified in the ParamNames
        Dim arrQueryStringParams
        arrQueryStringParams = Split(Session("ParamNames"), ";", -1, 1)

        ' get the number of parameters from the report
        Session("ParamCount") = Session("ParamsObject").Count

        ' construct the parameter dialog based on the information in the ParamsObject
        Dim nCurrTop
        Dim sDlgOuterHTML
        nCurrTop = 15
        nCtrlHeight = 30
        Session("DlgHeight") = 0     ' reset DlgHeight
        nQSUnspecifiedCount = 0

        For I = 0 To Session("ParamsObject").Count - 1
                nCtrlHeight = 30
                ' get the I-th parameter from the parameter collection
                Set oParam = Session("ParamsObject").Item(I)
                If Not IsInQueryStringSpecified(oParam, arrQueryStringParams) Then
                        ' label
                        sDlgOuterHTML = sDlgOuterHTML & GetLabelOuterHtml(oParam, nCurrTop)

                        ' get parameter's ArraySize
                        nArraySize = oParam.ArraySize

                        ' array of similar controls with no valid values list
                        If nArraySize > 1 And IsNull(oParam.ValidValues) Then
                                For J = 0 To nArraySize - 1
                                        sDlgOuterHTML = sDlgOuterHTML &
                                                GetCtrlOuterHtml(oParam, nCurrTop, I, J)
                                        nCurrTop = nCurrTop + 30
                                Next
                                nCtrlHeight = 0
                        ' list-Box
                        ElseIf nArraySize > 1 And oParam.MultiValue Then
                                sDlgOuterHTML = sDlgOuterHTML &
                                                GetListCtrlOuterHtml(oParam, nCurrTop, I)
                                nCtrlHeight = 80
                        ' combo-box
                        ElseIf Not IsNull(oParam.ValidValues) Then
                                sDlgOuterHTML = sDlgOuterHTML &
                                                GetComboCtrlOuterHtml(oParam, nCurrTop, I)
                        ' single control
                        Else
                                sDlgOuterHTML = sDlgOuterHTML &
                                                GetCtrlOuterHtml( oParam, nCurrTop, I, 0)
                        End If
                        nCurrTop = nCurrTop + nCtrlHeight
                        nQSUnspecifiedCount = nQSUnspecifiedCount + 1
                End If
        Next

        ' It is assumed that at least one parameter is explicitly not specified
```

```
            ' in order for the parameter dialog to prompt
            If nQSUnspecifiedCount > 0 Then
                    ' prompt the dynamically created parameter dialog
                    ' for the user to enter the required report parameters
                    Session("ShowParamsDialog") = 1

                    ' add Run and Cancel buttons to dialog content
                    sDlgOuterHTML = sDlgOuterHTML & _
                                    "<INPUT id=btnRun type=button value=Run name=btnRun
style='BACKGROUND-COLOR:gainsboro;Z-INDEX: 100; LEFT: 137px; WIDTH: 70px; FONT-FAMILY: sans-
serif; POSITION: absolute; TOP:" & CStr(nCurrTop + 10) & "px' width='70' LANGUAGE=javascript
onclick='return fnCloseDialog(1)'>" &
                                    "<INPUT id=btnCancel type=button value=Cancel name=btnCancel
style='BACKGROUND-COLOR:gainsboro;Z-INDEX: 101; LEFT: 215px; WIDTH: 70px; POSITION: absolute;
TOP:" & CStr(nCurrTop + 10) & "px' width='70' LANGUAGE=javascript onclick='return
fnCloseDialog(0)'>"

                    ' pass the dynamically built content of the parameter dialog
                    ' to the session variable
                    Session("DlgContent") = sDlgOuterHTML
                    Session("DlgHeight") = nCurrTop + 30
            Else
                    ' if otherwise the parameters are given do not prompt the parameter dialog,
                    ' simply pass these values to the global session variables
                    Session("ShowParamsDialog") = 0
            End If
    End If
End If
```

This code essentially begins by checking if *ReportEngine* and *Report* session variables are defined and set to valid objects. If the request action is "re-run" then the collection of parameters are retrieved via the *GetParams()* API function of the *ELSReport* class. Otherwise, the old instance of the *Report* object is destroyed and a new instance is created.

The next step is to define the connection string for the data access. In particular, if the *DBIndex* parameter is not specified in the *HREF* attribute of the calling HTML anchor, then the *ConnectString* is defined by the *ConnStr0* session variable evaluated in the Global.asa module. Otherwise, the proper *ConnStrJ* session value is used based on the *DBIndex* parameter's value, where (i.e. *J* = *DBIndex*). The resulting connection string value is then passed to the *ConnectEngine* function, which briefly speaking, defines and opens the data access connection, and then passes this connection to the report engine instance. Later on in this section, we will describe the details of the *ConnectEngine* function. For now, we will continue the outline of the remaining steps of the main section of the ReportViewer.asp ASP code.

The next step proceeds to define the *sBaseAddr* and *sCabAddr*. The later variable defines the path and filename of the PhookCtl.CAB, which contains the *Printer* extension class. The download and installation of this printer class is automated on demand via a Java script call, as we will see later on in the ReportViewer.asp code.

The process continues by calling the *LoadReport* function, which simply sets the *ReportFilename* property of the *ELSReport* object instance defined by the *Report* session variable. Note that in the *LoadReport* function the *RepFile* session parameter's value is used to evaluate this *ReportFilename* property. A closer observation will also reveal that along the filename of the *REP*-file, the value of the *RepFile* parameter must contain the relative subdirectory path with the prefix "`/`" character (e.g. see the *HREF* attribute values of the HTML anchors in the MainPage.htm file).

Given that loading the report file was successful, the process continues by evaluating the *RepName* session variable, and then using the *GetParams2* report engine API function retrieves the parameter options from the *REP*-file into the *ParamsObject* session variable. Also, if the submit action is not "`re-run`" (i.e. not a re-run of the same report) then the *HREF*'s *ParamNames* and *ParamValues* values are passed to the corresponding session variables. The content of the *ParamNames* are parsed and split into the *arrQueryStringParams* array, which will be used to check if any of the report parameters are explicitly specified in the anchor reference URI. Also, the *Count* property of *ELSParams2* object is called and passed to the *ParamCount* session variable, so that all the ingredients become ready for iteration on the report's parameter collection.

This iteration starts by setting the *oParam* object variable to the *I*-th item of the report parameter via the *Item* API

function. Then the process checks if the parameter is not explicitly specified in the URI of the MainPage.htm HTML anchor for the current report. At this point, observe that the iteration contains calls to the following helper functions:

GetLabelOuterHtml	to construct the DHTML code for the label control for the parameter. Note that the label text is retrieved from the *Prompt* property of the *ELSParamOption* object
GetComboCtrlOuterHtm	to construct the DHTML code for the combo-box control for the parameter's valid values, the content of the combo-box are constructed via the *GetOptionsString* helper function.
GetListCtrlOuterHtml	to construct the DHTML code for the list-box control for the parameter's valid values, the content of the list-box are constructed via the *GetOptionsString* helper function.
GetCtrlOuterHtml	to construct the DHTML code for other types of controls based on the data type of the parameter (note that this includes the calendar control implemented in Java script).

Therefore, the iteration dynamically constructs the controls for the report parameters, first constructing the label, then based on the *ArraySize* and other properties of the parameter, branches into the following conditionals:

Case 1: In this case it is assumed that we have an array of independent parameters. For each of these parameters the *GetCtrlOuterHtml* helper function is used to construct the DHTML code of the control.

Case 2: In this case it is assumed that we have a multi-valued parameter. Therefore a list-box must be used for the valid values list display. The *GetListCtrlOuterHtml* helper function is used to construct the DHTM: code of the control. Observe that in this case a height of 80 pixels is used for the height of this list-box control.

Case 3: In this case it is assumed that we have a parameter with valid values list. Therefore, a combo-box must be used for the valid values list display, and the DHTML code is constructed via the *GetComboCtrlOuterHtml* helper function.

Case 4: Finally, the remaining case is when the parameter requires a single control with no valid values list. For this kind of single control, the *GetCtrlOuterHtml* helper function is used to construct the corresponding DHTML code.

The logic at this point is as follows. If at least one report parameter is unspecified in the calling URI of the HTML anchor for the report, then the parameter dialog should prompt for user input. Otherwise, the parameter values are directly passed to the session variables and the *ShowParamsDialog* is set to 0, so that no parameter dialog is prompt.

A glance at the details of the construction of parameter controls in the *GetCtrlOuterHtml*, *GetListCtrlOuterHtml* and *GetComboCtrlOuterHtml* functions reveals the usage of the *Prompt*, *AllowBlank*, *Nullable*, *DataType* and *Value* properties of the *ELSParamOption* object. Moreover, for parameters with valid values lists, we have utilized the *GetOptionsString* function in the construction of the *GetListCtrlOuterHtml* and *GetComboCtrlOuterHtml* functions. The following is a code listing of the *GetOptionsString* function:

```
Function GetOptionsString(collValidVals, collDefVals, nInx, bMulti)
    nOptionCount = 0
    nSelCount = 0
    ' get count of valid values
    If IsObject(collValidVals) Then
           nOptionCount = collValidVals.Count
    End If
    ' get count of default values
    If IsObject(collDefVals) Then
           nSelCount = collDefVals.Count
    End If
    Dim sResult
    ' loop over the valid values and possible default values constructing the lists
    For k = 0 To nOptionCount - 1
           bSelected = False
           Set oOption = collValidVals.Item(k)
```

```
            If Request.QueryString("Action") <> "re-run" Then
                  If Not bMulti Then
                        If oOption.Value = collDefVals.Item(0).Value Then
                              bSelected = True
                        End If
                  Else
                        For l = 0 To nSelCount - 1
                              Set oSel = collDefVals.Item(l)
                              If oOption.Value = oSel.Value Then
                                    bSelected = True
                                    Exit For
                              End If
                        Next
                  End If
            ElseIf IsObject(oOldParams) Then
                  If Not bMulti Then
                        If oOption.Value = oOldParams.Item(nInx).Value(0) Then
                              bSelected = True
                        End If
                  Else
                        nSelCount = oOldParams.Item(nInx).ArraySize
                        For b = 0 To nSelCount - 1
                              If oOption.Value = oOldParams.Item(nInx).Value(b) Then
                                    bSelected = True
                                    Exit For
                              End If
                        Next
                  End If
            End If
            If bSelected Then
                  sResult = sResult & "<OPTION VALUE='" & oOption.Value &
                                          "' SELECTED>" & oOption.Name
            Else
                  sResult = sResult & "<OPTION VALUE='" & oOption.Value & _
                                          "'>" & oOption.Name
            End If
      Next
      GetOptionsString = sResult
End Function
```

We next cover the details of the *ConnectEngine* function, which has the following code listing:

```
Function ConnectEngine (sConnString)
      On Error Resume Next
      ConnectEngine = False

      If Session("EngineConnected") Then
            ConnectEngine = True
      Else
            ' create an instance of connection object
            Set Session("Conn") = server.CreateObject("ADODB.Connection")
            If Err.number <> 0 Then
                  Response.Write "Failed to create ADODB Connection object on the server."
                  Err.Clear
            Else
                  If IsObject(Session("Conn")) Then
                        ' open the connection
                        Session("Conn").Open sConnString, "", "", adOpenUnspecified

                        If Err.Number <> 0 Then
                              Response.Write "Failed to find database on the server."
                              Session.Abandon
                        Else
                              ' pass the connection object to the report engine
                              If Session("Conn").State = adStateOpen Then
                                    Session("Conn").CursorLocation = adUseClient
                                    Session("ReportEngine").Connection = Session("Conn")
                                    ConnectEngine = True
                              End if
                        End If
                  End If
            End If
```

```
    End If
End Function
```

At the end of the HTML <HEAD>-section, the Java script *window_onload()* function is executed on the *onload* event of the script. This function calls the *fnIsAXLoaded()* function, which basically loads the phookctl.Print class. At this point, if the Phookctl.CAB is not already downloaded and installed on the client machine, the process automatically downloads and installs the *CAB* file. The rest of the code in the ReportViewer.asp contains print settings, preview and other printer template related functions. These functions essentially call respective functions of the installed *Print* class.

This completes the dynamic parameter dialog construction. Note that the ReportViewer.asp page has an HTML part that consists of a frameset containing two HTML frames, respectively with *SRC* attribute values "RGOutput.htm" and "RGController.asp". In particular, the execution of the ReportViewer.asp page will in turn execute the RGController.asp page, which we will consider next.

The loading of the RGController.asp page triggers the *window_onload* Java script function defined as follows:

```
function window onload() {
    nCurrPage = 1;
    nPageCount = 1;

    frmParams.btnStop.disabled = true;

    frmParams.btnFirst.disabled = true;
    frmParams.btnPrev.disabled = true;
    frmParams.btnNext.disabled = true;
    frmParams.btnLast.disabled = true;
    frmParams.btnMPrintCancel.disabled = true;

    if(fnEnterParams())
            fnSubmitForm("run");
}
```

This function initializes variables and disables some buttons, and then calls the *fnEnterParams* function, which has the following code:

```
function fnEnterParams()
{
    // check if the ShowParamsDialog session variable is true
    sText = document.getElementById('ShowParamsDialog').innerText;
    var nShowDlg = parseInt(sText);
    if(nShowDlg == 1)
    {
            // get the value of the DlgHeight session variable
            sText = document.getElementById('paramdlgheight').innerText;
            var nDlgHeight = parseInt(sText);
            if(nDlgHeight > 0)
            {
                    // get the value of the DlgContent session variable set in ReportViewer.asp
                    var vArgs = new Array(2);
                    vArgs[0] = document.getElementById('dlgContent').innerHTML;
                    // compute the dialog height
                    dlgHeight = Math.min(nDlgHeight + 50, 500);
                    // call the ParametersDlg.htm page as a dialog, passing the DlgContent
                    // via the vArgs array, and the calculated height via dlgHeight
                    nResult = showModalDialog("ParametersDlg.htm", vArgs,
                            "center=yes;help=no;dialogWidth='420px';status=no; dialogHeight='"
                            + dlgHeight +"px'" );
                    if (nResult == 1)
                    {
                            // the dialog returns vArgs array with the parameter names and values,
                            // these values are passed to hidden variables of frmParams form
                            // (check the end of this file for the hidden variables)
                            frmParams.ParamName.value = vArgs[0];
                            frmParams.ParamValue.value = vArgs[1];
                    }
```

```
                    delete vArgs;
                    return nResult == 1;
                }
            }
        return true;
}
```

This code essentially calls the ParametersDlg.htm as parameters dialog passing the dialog content already constructed in the ReportViewer.asp. Note that when this parameters dialog is submitted, the `vArgs` array is returned with `vArgs[0]` consisting of a sequence of the parameter label names separated by "`;`" character, and similarly the `vArgs[1]` consisting of a sequence of entry values separated by "`;`" character. This parameter information is passed to the hidden *ParamName* and *ParamValue* elements of the *frmParams* form. Incidentally, the *frmParams* form is defined at the end of the RGController.asp file, and has an *action* attribute value of "`RepGen.asp`", hidden elements *Action*, *ParamName* and *ParamValue*, and several toolbar buttons with command functions described next.

Figure 3.6. Showing the web toolbar buttons of the RGController.asp page

Figure 3.6 shows the *frmParams* form's buttons that are displayed at the bottom of the report viewer page. These buttons are defined in the HTML `<BODY>`-section of the RGController.asp page via `<INPUT>` elements (look at the end of the file for more details). The following table lists these buttons from left to right along with their corresponding functionality:

Button Name	Called Function	Action Description
btnRun	btnRun_onclick	This command will re-run the report
btnStop	btnStop_onclick	This command will attempt to halt an active report generation process
btnFirst	btnFirst_onclick	This command will display the first page of the generated report output in the report viewer
btnPrev	btnPrev_onclick	This command will display the previous page of the generated report output in the report viewer
btnNext	btnNext_onclick	This command will display the next page of the generated report output in the report viewer
btnLast	btnLast_onclick	This command will display the last page of the generated report output in the report viewer
btnPreview	btnPreview_onclick	This command will print preview the current page of the generated report output
btnPrint1	btnPrint1_onclick	This command will print the current page of the generated report output
btnMPrint	btnMPrint_onclick	This command will call the Print dialog to print a range of the pages from the generated report output
btnMPrintCancel	btnMPrintCancel_onclick	This command will cancel an active print process
btnSaveAs	btnSaveAs_onclick	This command will download the whole report output as a single HTML document and display it in a new IE browser window to be saved or exported on the client machine
btnMainWindow	btnMainWindow_onclick	This command will navigate back to the MainPage.htm page

Getting back to our outline of the RGController.asp module, observe that once the parameter dialog is submitted, the *fnSubmitForm* function is called displaying status message via cookies and passing argument "`Run`" via hidden *Action* variable to the RepGen.asp ASP page. Therefore, we will describe next the processes involved in the RepGen.asp module, and then get back to the RGController.asp module to outline more details about the various button commands.

The following code contains the main APS processes of the RepGen.asp ASP page:

```
On Error Resume Next

' if the action is the initial execution of a report or the re-run via the Run button
' of the report viewer page, then initialize the report and call the GenerateReport function
If Request.Form("Action") = "run" Or Request.Form("Action") = "re-run" Then
    If Request.Form("Action") = "re-run" Then
        Response.Redirect "ReportViewer.asp?RepFile=" & Session("RepName") & "&Action=re-run"
    End If
    ' initialize and prepare the report
    Session("ReportInitialized") = InitReportInfo
    If Session("ReportInitialized") Then
        ' generate the report output
        GenerateReport
    End If
ElseIf Request.Form("Action") = "stop" And Session("InGeneration") = 1 Then
    ' if a generation process is active then call the Stop API
    Session("ReportEngine").Stop
    Response.Write "Report generation is terminated by user request."
Else
    Select Case Request.Form("Action")
        Case "first"
            Session("CurrPage") = 1
        Case "prev"
            If Session("CurrPage") > 1 Then
                Session("CurrPage") = Session("CurrPage") - 1
            End If
        Case "next"
            If Session("CurrPage") < Session("PageCount") Then
                Session("CurrPage") = Session("CurrPage") + 1
            End If
        Case "last"
            Session("CurrPage") = Session("PageCount")
    End Select
    ' get respective page content and pass as response along with print settings info
    sContent = Session("ReportEngine").GetPageContent(Session("CurrPage"))
    If Err.Number <> 0 Then
        Response.Write "Failed to obtain the " & Session("CurrPage") & " page of report."
        Err.Clear
    Else
        Response.Write sContent
        Response.Write GetPrintSettings()
    End If
End If
```

The RepGen.asp page starts with checking the request form's *Action* value. If this value is either a "`run`" or "`re-run`", then it proceeds to the actual generation of the report output. Recall that the "`re-run`" *Action* value occurs when the user in the report viewer (most probably after a report generation) clicks on the **Run** button. Otherwise, the currently selected report is being generated the first time from the MainPage.htm URI link. In this case, first, the report is initialized by resetting session variables and evaluating the parameters of the report via the helper function *InitReportInfo*, and then the report generation is triggered via the *GenerateReport* function. We will consider next the details of these two helper functions.

First, we will look at what goes on in the *InitReportInfo* function, which has the following code listing:

```
Function InitReportInfo
    If Session("ParamCount") > 0 Then
        ' split report parameter names specified in the QueryString
        Dim arrQueryStringParamNames
        arrQueryStringParamNames = Split(Session("ParamNames"), ";", -1, 1)
        ' split report parameter values specified in the QueryString
        Dim arrQueryStringParamValues
        arrQueryStringParamValues = Split(Session("ParamValues"), ";", -1, 1)
        ' get the values of the remaining parameters specified in Parameters dialog
        Dim arrDlgParamValues
        arrDlgParamValues = Split(Request.Form("ParamValue"), ";", -1, 1)
        ' define offset to take account for the case of parameter arrays
        ' among the parameters shown in the dialog
```

```
        offsCtlArr = 0
        ' count of parameters already specified through QueryString
        nQSSpecifiedCount = 0
        ' set report parameter values
        For I = 0 To Session("ParamCount") - 1
           Set oParam = Session("ParamsObject").Item(I)
           nValueArrSize = oParam.ArraySize
           bSuccess = true
           nInx = GetQueryStringParamIndex(oParam, arrQueryStringParamNames)
           If nInx >= 0 Then
               ' the value is specified in the QueryString
               If UBound(arrQueryStringParamValues) >= nInx Then
                   'the corresponding value exists
                   bSuccess = SetParameterValue(oParam, arrQueryStringParamValues(nInx), 0)
               End If
               nQSSpecifiedCount = nQSSpecifiedCount + 1
           Else
               ' the value is specified through the parameters dialog
               If nValueArrSize > 1 And IsNull(oParam.ValidValues) Then
                   ' array parameter in the parameters dialog
                   For F = 0 To nValueArrSize - 1
                     If F > 0 Then
                            offsCtlArr = offsCtlArr + 1
                     End If
                     bSuccess = SetParameterValue(oParam,
                                 arrDlgParamValues(I + offsCtlArr - nQSSpecifiedCount), F)
                     If Err.Number <> 0 Then
                            Response.Write "Failed to set " & oParam.Ptompt & _
                                        " parameter value." & vbCrLf & Err.Description
                            Err.Clear
                            InitReportInfo = False
                            Exit Function
                     End If
                   Next
               Else
                   bSuccess = SetParameterValue(oParam,
                                 arrDlgParamValues(I + offsCtlArr - nQSSpecifiedCount), 0)
               End If
           End If
           If Not bSuccess Then
               InitReportInfo = False
               Exit Function
           End If
        Next
    End If
    ' find and pass the target browsers URL via the SetBaseURLPath API function of report engine
    sBaseAddr = "http://" & Request.ServerVariables("SERVER_NAME") & _
                                    Request.ServerVariables("PATH_INFO")
    ' pass the base URL for multi-page print support via the SetBaseMPrintSupportURL API
    Session("ReportEngine").SetBaseMPrintSupportURL Left(sBaseAddr,
                                    InStrRev(sBaseAddr, "/", -1, 1)) & "BIN/"
    sRepPath = Session("RepName")
    sRepPath = Left(sRepPath, InStrRev(sRepPath, "/", -1, 1))
    sBaseAddr = Left(sBaseAddr,InStrRev(sBaseAddr, "/", -1, 1)) & sRepPath

    Session("ReportEngine").SetBaseURLPath sBaseAddr
    InitReportInfo = True
End Function
```

This function essentially begins by parsing and evaluating the parameters passed from the MainPage.htm page, as well as the parameters from the parameters dialog. Special consideration is given to array parameters coming from multi-valued report parameters. Then the printer template base URL and the report's dependencies base URL are constructed and passed to the report engine via the *SetBaseMPrintSupportURL* and *SetBaseURLPath* API functions.

Next, we will look at the *GenerateReport* function, which has the following code listing:

```
Sub GenerateReport
    On Error Resume next
    Session("InGeneration") = 1
    lRes = -1
```

```
    Session("PageCount") = 1

    ' call GenerateReport API of the report engine passing the report object as argument
    lRes = Session("ReportEngine").GenerateReport(Session("Report"), ELS HIDE)
    If lRes < 0 Then
           Response.Write "Generation Error: " & Err.Description
           Err.Clear
           Session.Abandon
           Exit Sub
    End If
    Session("InGeneration") = 0
    Response.Cookies("PageCount") = lRes
    Session("PageCount") = lRes
    ' after successful generation get the content of the first page of the output
    ' via the GetPageContent API function of the report engine
    If lRes = 1 Then
           sContent = Session("ReportEngine").GetOutput
    Else
           sContent = Session("ReportEngine").GetPageContent(1)
    End If
    If Err.Number <> 0 Then
           Response.Write "Failed to obtain the fisrt page of " & _
                                  Session("Report").ReportFileName & "."
           Err.Clear
    Else
           Session("CurrPage") = 1
           ' return the content of the output page to the web browser
           ' also return the default print settings of the report hidden in a <DIV>-element
           Response.Write sContent
           Response.Write GetPrintSettings()
    End If
End Sub
```

The *GenerateReport* subroutine essentially starts with initializing some variables then calls the *GenerateReport* API function of the report engine passing the *Report* object as an argument. After successful generation of the report output, the first page is retrieved via the *GetPageContent* API function returning the content of the page as response along with the printer settings information via a hidden `<DIV>`-element.

We are ready now to get back to the RGController.asp and outline the details of the button commands of the *frmParams* form.
First, note that the *btnRun* and *btnStop* command buttons will trigger the RegGen.asp execution with *Action* values respectively "`re-run`" and "`stop`". The *re-run* action will restart the ReportViewer.asp call with the additional *Action* parameter, which will essentially maintain the old parameter information and repeat all the report processes that we have outlined so far. The *stop* action on the other hand, will halt any active report generation process.

The next four buttons *btnFirst*, *btnPrev*, *btnNext* and *btnLast*, will result into RegGen.asp execution with respective *Action* values "`first`", "`prev`", "`next`" and "`last`". This will call on the *GetPageContent* API function of the report engine to display the respective pages of the report output in the report viewer frame.

The *btnPreview* and *btnPrint1* buttons respectively call the *fnPreview_onclick* and *fnPrint1_onclick* functions. These functions in turn call the *fnPrintPreview* and *fnPrintOnePage* functions of the ReportViewer.asp, which in turn call the *PrintPreview* and *PrintCurrPage* functions of the *Print* class (assuming that the PHookCtl.dll was automatically installed via the Phookctl.CAB file).

The *btnSaveAs* button will simple open a new instance of the IE browser activating the SaveOutput.asp page, with the query string "`SaveOutput.asp?nPrint=1`". The entire ASP code of the SaveOutput.asp page is listed below:

```
If IsObject(Session("ReportEngine")) Then
    nPrint = Request.QueryString("nPrint")
    If nPrint = 1 Then
        lRres = Session("ReportEngine").SendOutputHTTP(Response, , , 1)
        If lRes < 1 Then
           Response.Write "SendOutputHTTP returns: " & lRes
        End If
    ElseIf nPrint = 2 Then
        Session("ReportEngine").SendOutputHTTP Response,
```

```
                        Request.QueryString("nStart"), Request.QueryString("nEnd"), 2
    End If
End If
```

In particular, since in this case the *nPrint* is equal to 1, the *SendOutputHTTP* API function is called with *lReserved* argument set to 1. This indicates that the whole report output must be downloaded to the client side, using the HTTP 1.1 compression protocol.

Finally, the *btnMPrint* will essentially call the print dialog, so that the user may select a range of pages from the report output to be printed on the client side. The details of this command, as well as that of the *btnMPrintCancel* command will be described in the technical article with the "Affordable Web Reporting Solution: Part II" title.