2

The SCRIPT Language

The ELS-Script® language, also referred to as the **SCRIPT**, is essentially a powerful scripting language specially designed for the purpose of the creation of standard, as well as complex report documents.
It defines a full set of syntax elements destined to introduce a new direction in the technology of modern report making. Already, the traditional snapshot based report technology is becoming outdated with the rise of the Internet and the globalization of several markup language standards, such as HTML, DHTML, VB Script, Java Script, ASP, CSS, XML and XSLT. As an analogy, we may say with great confidence that this new report technology introduced by the SCRIPT is to the traditional old report technology, as the .NET web development environment is to the ASP, VB scripting and Java scripting technology. In other words, it is a remarkable combination of the best of all worlds.

The features of the SCRIPT language cover not only the presentation, but all aspects of report development cycles, from the query parameter selection to the data binding; the generation and the presentation of the report's result output in standard open document format. It has the capabilities to produce literally any kind of multi-section, multi-group, cross tabular, record or matrix based report, or for that matter, any presentation document. In particular, such documents may be HTML formatted reports, web pages, Active Server Pages, XML files, or any markup language based document.

The prerequisites for this chapter are kept to the minimum. In particular, some basic knowledge of SQL and HTML is required. An outline of all HTML elements that are important in the context of the SCRIPT language will be given. In addition, the reader must have some prior working knowledge of software programming. To acquire experience with the SCRIPT language, it is recommended to download a trial version, or purchase the release version of the ELS-Script® package. The package comes with a stand-alone visual report designer, a script editor, the report compiler along with the report engine (the product may be purchased from Epsilon-Logic Systems at the following URL: http://www.epsilon-logic.net/ELSScript.asp).

The current implementation of the SCRIPT language is specialized to XHTML formatted report output generation, which naturally is very suitable for the application of web reporting. This chapter therefore will cover the description of all elements of the syntax of the SCRIPT language with simple, to moderate, illustrations and code samples based on XHTML format.

In brief, we will discuss the following issues:

- Advantages in designing reports using the SCRIPT language verses the visual methods
- Basic and advance syntax elements of the SCRIPT language, with step by step tutorials and hands on code illustrations
- Utilizing the SCRIPT editor that comes with the ELS-Script package
- Application of the SCRIPT to some real-world report examples, where we will give a complete and self-contained coverage of the design of sample reports, as well as, the underlying sample database

construction

- Overview of the basic architecture of the report SCRIPT compiler and output generator

Why to SCRIPT?

Yes, why to SCRIPT? When all the report tool manufacturers on the market are advertising that theirs is the tool with ingenious visual report designer that requires absolutely no complicated coding or scripting. The current state of the art of the report making has traditionally evolved along the collective pattern of almost 20 years of software technology hype that has created the surprising baseless demand, that modern report tools must have absolutely no scripting capabilities.

The story goes as follows. Back in the 80's software and computer industry giants such as Apples, Microsoft, HP, IBM and others, were in for a race to develop the most graphical or visual user interfaces. Back then, some terms such as WYSIWYG (acronym of *What You See Is What You Get*) played a very powerful role in the daily tasks of the average office employee, whose computer operation skills were extremely limited. With the advent of the TrueType fonts and laser printer technologies, WYSIWYG became the norm for document creation tool repertoire. As a result the concept of device independence became feasible, so that for any selected printer the corresponding device context maps the document page to the display monitor (e.g. in the print preview), approximating the appearance of the actual printed page with high degree of precision.

Report tool developers incorporated these WYSIWYG features to simplify the creation and design of continuous report forms, essentially bringing the art of report making down to the level of average office employee's computer skills. There approach was statistical in nature and had the following primary goals:

- To maximize visual design methodology in designing a report
- To eliminate coding or scripting from the report tool
- To simplify report making process at least for most standard report types

Although none of these conditions mutually exclude each other, it was mysteriously assumed that visual design methods should eliminate scripting, while simplification should rule out advance enhancements or extensibility. Not to mention the little old golden rule of reusability. Therefore, report making problems were solved in a quantitative rather than qualitative manner, which certainly satisfied 60-70% of most standard report types.

One does not have to think much to understand that to support full scripting capabilities in a report tool does not necessarily imply that it should not have visual designer. Nor simplification implies sacrificing enhanced functionality and extensibility. What about the "best of both worlds" motto? Obviously the original developers of the existing report tools in the market, tried to avoid writing parsers and compilers.

Therefore, our initiative in the development of ELS-Script® report tool had the following goals:

- To maximize visual design methodology in designing a report
- To provide full scripting capability to the report designer
- To simplify the tasks of report making as well as data retrieval
- To enhance functionality to cover virtually any data representation in any medium
- To define a simple and universal report template format for standardization of industry level report types
- To streamline the art of report making with the current software technologies

Advantages of using the SCRIPT

The art of report making must incorporate both powerful visual design methods and scripting, moreover this scripting must be complete, covering all aspects of the report template and therefore serving as a source code. Most third party report tools satisfy this condition only partially, while ELS-Script® report tool is designed ground

up to fully satisfy this condition.
To illustrate the importance of scripting we will outline next the major tasks that must be undertaken for the preparation of a standard report. After acquiring the user requirements:

1. Analysis of the data and report content must be performed.
2. The actual report page must be designed, which incorporates constant text labels, graphic elements, variable sections, page setup, etc.
3. Query commands must be built, to define the field elements that will be included in the variable sections.
4. Fields and formulas must be included.
5. Grouping, aggregations and sorting must be defined.
6. Query parameter form and report generator integration issues must be addressed in the host applications.

Of these only the second task is where most of the visual design methods can be applied, while all the other tasks require coding or scripting to a certain extend. In fact, even in the second task there are operations that may be simplified if scripting is incorporated side by side with visual design methods. For example, positioning borders, lines, text and graphic objects, via free-style pointer device maneuvers is much more difficult and inaccurate, than simply entering precise coordinate values relative to the page margins. In contrast, the other tasks such as query building, formula or expression building, as well as defining nested or hierarchical groups, are in essence so natural when accomplished via scripting.

The SCRIPT language supports all aspects of report development tasks, for example there are special syntax elements that handle page setup and other report setting parameters. It has full conditional controls and iteration elements to define grouping and complex data representation. Finally, it has special syntax elements that can define any query parameter form, so that the complicated task of integration of the report generator engine with the host application can be achieved by adding just a few lines of code.

In ELS-Script® report tool, the contents of a report file are defined and stored via the SCRIPT language. This assures full support for maintenance, as well as reusability of report files as templates. The language features cover *Standard Report Templates*, which combined with preprocessor directives increases the reusability index to maximum efficiency.

Basics of the SCRIPT

The SCRIPT language is a mixture of SQL procedural language combined with the syntax and the flavor of the Visual Basic function attributes. The reason behind the SQL procedural nature of the SCRIPT language is two fold. First, it provides a unified simple language for both data retrieval as well as data representation. For example, to build queries one needs some familiarity with the SQL statement language, moreover to write stored-procedures additional knowledge of SQL procedural language is required. And therefore, instead of escalating the learning curve for users, by introducing new language syntax, we have used a syntax which is very similar to SQL procedural language.
The second reason is that often the advance reports are developed by SQL programmers or database administrators, and therefore, defining the SCRIPT language in their native language will definitely increase the popularity and utilization of the ELS-Script® software system.

In the following we will define the basic syntax elements of the SCRIPT language; essentially covering:

- HTML and ELS tags
- Report sections
- Basic data types
- Declaration and evaluation
- Operations on data types
- Conditional and iteration controls

HTML and ELS Tags

The concept of the SCRIPT language evolved around the markup language scripting; very similar to VB, Java or ASP scripting. Essentially the HTML code would be mixed with these script elements distinguished via tag identifiers. But unlike VB, Java or ASP scripting, the SCRIPT language is not intended to be an instruction list for a virtual machine running as a background server. Rather, it is first compiled into an object form, which then is served as specifications when the report engine is called, to generate the output of the report.
Therefore, to proceed in the description of the SCRIPT language, we need some working knowledge of HTML coding. Note though that the ELS-Script® package contains a visual HTML designer, and that it is not a requirement to be an HTML guru to be able to use ELS-Script® report tool.
The following table gives an overview of all the HTML tags with attributes that are relevant to the SCRIPT language:

Elements	Description
<!-- , -->	Comments (start and end tags)
<B>	Bold text
<BASEFONT>	Default font for a document section Relevant attributes: COLOR, FACE, SIZE, etc.
<BIG>	Big font relative to the current font
<BODY>	The document displayable content Relevant attributes: BACKGROUND, BGCOLOR, BGPROPERTIES, BOTTOMMARGIN, LEFTMARGIN, RIGHTMARGIN, TOPMARGIN, TEXT, SCROLL, etc.
 	Line break
<CAPTION> <CENTER>	Figure or table caption Center the alignment
<COL>	Column in COLGROUP or TABLE Relevant attributes: ALIGN, CHAR, CHAROFF, SPAN, VALIGN, WIDTH, etc.
<COLGROUP>	Column group in a TABLE Relevant attributes: ALIGN, CHAR, CHAROFF, SPAN, VALIGN, WIDTH, etc.
<DIV>	A block of document content treated as a logical unit Relevant attributes: ALIGN, CHARSET, etc.
<EM>	Emphasized text
<EMBED>	Embedded object Relevant attributes: (all attributes)
<FONT>	Font specification Relevant attributes: COLOR, EFFECT, FACE, SIZE, WEIGHT, etc.
<FRAME>	HTML frame used with FRAMESET Relevant attributes: BORDERCOLOR, FRAMEBORDER, MARGINHEIGHT, MARGINWIDTH, NORESIZE, SCROLLING, etc.
<FRAMESET>	Used with FRAME Relevant attributes: BORDER, BORDERCOLOR, COLS, FRAMEBORDER, FRAMESPACING, ROWS, etc.
<H1>, ..., <H6>	Heading levels 1 through 6
<HR>	Horizontal rule Relevant attributes: ALIGN, COLOR, INVERTBORDER, NOSHADE, SIZE, WIDTH, etc.
<I>	Italic text
<IFRAME>	Independently controllable content region Relevant attributes: (all attributes)
<IMG>	Graphic image Relevant attributes: (all attributes)

<MULTICOL>	Multiple column text Relevant attributes: COLS, OUTTER, WIDTH, etc.
<NOBR>	No breaks
<OBJECT>	Object embedding Relevant attributes: (all attributes)
<P>	Paragraph section Relevant attributes: ALIGN, etc.
<PARAM>	Parameter for OBJECT
<PLAINTEXT>	Plain text
<PRE>	Preformatted text, preserves all characters (e.g. many spaces)
<S>	Strike through text
<SCRIPT>	Scripting
<SMALL>	Small text relative to current font
<SPACER>	Extra space
<SPAN>	Text span
<STRIKE>	Strikeout text
<STRONG>	Strong emphasized text
<SUB>	Subscript text
<SUP>	Superscript text
<TABLE>	HTML table, this is very important in SCRIPT language Relevant attributes: ALIGN, BACKGROUND, BGCOLOR, BORDER, BORDERCOLOR, BORDERCOLORDARK, BORDERCOLORLIGHT, CELLBORDER, CELLPADDING, CELLSPACING, COLS, FRAME, HEIGHT, RULES, WIDTH, etc.
<TBODY>	Table body Relevant attributes: (all attributes)
<TD>	Table data (or cell) Relevant attributes: (all attributes)
<TFOOT>	Footer section of TABLE Relevant attributes: (all attributes)
<TH>	Table header Relevant attributes: (all attributes)
<THEAD>	Header section of TABLE Relevant attributes: (all attributes)
<TR>	Table row Relevant attributes: (all attributes)
<TT>	Teletype text
<U>	Underlined text

Table 2.1. *The HTML tags and attributes relevant to SCRIPT*

The SCRIPT compiler that is included in the ELS-Script® package, parses the HTML together with the special report script tags, and then processes all HTML material with respect to these SCRIPT instructions. In general, all report specific instructions must be included between the <ELS> and </ELS> tags. The short version <!> and </!> of this tags are also allowed, and may be used specially for SCRIPT element sections that fit on a single line. Also one may optionally use the HTML comment symbols <!-- and --> to hide SCRIPT elements from Internet browsers. In any case, these symbols are ignored by the SCRIPT compiler. The following code examples all have the same effect:

```
      ... HTML elements here...
<ELS>
      ... SCRIPT elements here ...
</ELS>
      ... HTML elements here ...
```

here is an alternative short tagged form of the above example:

```
      ... HTML elements here...
<!>
      ... SCRIPT elements here ...
</!>
      ... HTML elements here ...
```

which is the same as the following:

```
      ... HTML elements here...
<!> ... SCRIPT elements here ... </!>
      ... HTML elements here ...
```

here is the same code with the HTML comment symbols:

```
      ... HTML elements here...
<ELS>
<!--
      ... SCRIPT elements here ...
-->
</ELS>
      ... HTML elements here ...
```

Note that the SCRIPT elements between the HTML comment symbols are processed by the SCRIPT compiler, and that only the symbols `<!--` and `-->` are ignored (and not the text in between). In contrast to comment SCRIPT elements themselves one must use either of the following syntax (similar to C/C++ comment notation):

`//` to comment the line after the symbol,
`/* */` to comment multiple line between these two symbols.

Just like HTML tags, some SCRIPT tags have attributes, for example the `<ELS>` tag has the `NAME` attribute, which may be used to identify uniquely the tag. Here is a sample code of `<ELS>` tags with the `NAME` attribute:

```
      ... HTML elements here...
<ELS NAME="MyFirstTag">
      ... SCRIPT elements here ...
</ELS>
      ... HTML elements here ...
```

Report Sections

In the real industrial world, most standard reports consist of essentially the following sections:

Report Header which appears once at the beginning of the report

Page Header which appears once at the top of each page in the report

Report Detail which contains the data presentation section of the report

Page Footer which appears once at the bottom of each page in the report

Report Footer which appears once at the end of the report

Often, the *Report Header* contains some introductory material to the report document, such as: images, logos, charts, coverage information, etc. In some cases, this header section may also include report summaries or analyses that are usually included in the end of the report. In a symmetric way, the *Report Footer* may include concluding

material or summaries for the report. In contrast, the *Page Header* and *Page Footer*, may contain page specific material, such as page number, page count, section or group title, footnotes, date, reference, as well as, image and other such material. The *Report Detail* is where the body of the report should reside, which consists of data presentation in a continuous iterative manner, often using a tabular structure along with nested groups and categories.

The SCRIPT language has special tags for each of these traditional report sections along with a few other program segment sections. These program segment sections handle the report settings, the background stored-procedure instructions, the query parameter and the interface parameter lists, essentially rendering the report file to be self-contained and complete.
These tags are listed below:

`<ELS_QPARAM> ... </ELS_QPARAM>`	to include interface and query parameter instructions
`<ELS_OPROCS> ... </ELS_OPROCS>`	to include stored-procedure and other process calls
`<ELS_RSETTINGS> ... </ELS_RSETTINGS>`	to define the report settings
`<ELS_RHEADER> ... </ELS_RHEADER>`	to include the report header specifications
`<ELS_PHEADER> ... </ELS_PHEADER>`	to include the page header specifications
`<ELS_RDETAIL> ... </ELS_RDETAIL>`	to include the report detail specifications
`<ELS_PFOOTER> ... </ELS_PFOOTER>`	to include the page footer specifications
`<ELS_RFOOTER> ... </ELS_RFOOTER>`	to include the report footer specifications

The first three section tags allow only SCRIPT elements between the start and end tags, while the other section tags may contain SCRIPT elements intermixed with HTML or other scripts, depending on the document format. In general, when using SCRIPT elements in the mixed tag types, one must adhere to the `<ELS>` tag usage. The following is a generic example of code illustrating the usage for all section tags:

```
// generic usage of the report sections
#define LINK_PATH "THIS FILE\Include\OrderForm.css"
#include "Include\OrderForm.txt"

<ELS OPROCS>
     // call store-procedures here,
     // you can use only SCRIPT elements in this tags
</ELS OPROCS>

<HTML>
<HEAD>
<LINK href=LINK_PATH type=text/css rel=stylesheet></LINK>
<BODY>

<ELS_QPARAMS>
     // include the interface or query parameters here,
     // you can use only SCRIPT elements in this tags
</ELS_QPARAMS>

<ELS RSETTINGS>
     // include report settings here,
     // you can use only SCRIPT elements in this tags
</ELS RSETTINGS>

<ELS RHEADER>
     // include report header specifications here,
     // you can use both HTML and SCRIPT elements in this tags
</ELS RHEADER>

<ELS_PHEADER>
     // include page header specifications here,
     // you can use both HTML and SCRIPT elements in this tags
</ELS_PHEADER>
```

```
<ELS_RDETAIL>
     // include report details here,
     // you can use both HTML and SCRIPT elements in this tags
</ELS RDETAIL>

<ELS_PFOOTER>
     // include page footer specifications here,
     // you can use both HTML and SCRIPT elements in this tags
</ELS_PFOOTER>

<ELS_RFOOTER>
     // include report footer specifications here,
     // you can use both HTML and SCRIPT elements in this tags
</ELS_RFOOTER>

</BODY>
</HTML>
```

Note that the mixed language sections `<ELS_RHEADER>`, `<ELS_PHEADER>`, `<ELS_RDETAIL>`, `<ELS_PFOOTER>` and `<ELS_RFOOTER>` must be positioned within the HTML body of the report script, while the other sections may optionally be put inside or outside the HTML body.
We should emphasize that the page header and page footer sections are a little different than the others, in the sense that they possess the `HEIGHT` attribute to control the absolute height of the section in the report output.

Basic Data Types

The wealth of a programming language often is related to the completeness of the data type system that it supports. In the design of the SCRIPT language, data types were implemented with the following goals in mind:

1. They must be standard and already well established in most programming languages.
2. They must be interoperable with most backend database engines, in a sense very similar to the Common Type System of the .NET platform.
3. They must constitute a complete data type system.

In an attempt to realize these conditions, specially, since SCRIPT language has the SQL Server flavor, it was decided to adopt data types very similar to that of SQL Server 7. Therefore, the data types in the SCRIPT language are as follows:

`bit`	integer data with either 1, 0 or NULL value.
`int`	4-byte integer data type, ranging from -2^{31} (-2,147,483,648) through $2^{31} - 1$ (2,147,483,647).
`smallint`	2-byte integer data type, ranging from -2^{15} (-32,768) through $2^{15} - 1$ (32,767).
`tinyint`	1-byte integer data type, ranging from 0 through 255.
`numeric`	numeric data type, ranging from $-10^{28} + 1$ through $10^{28} - 1$, with a maximum precision of 15 digits.
`money`	monetary data type, ranging from -2^{63} (-922,337,203,685,477.5808) through $2^{63} - 1$ (922,337,203,685,477.5807), with accuracy to ten-thousandth of a monetary unit.
`smallmoney`	monetary data type, ranging from -2^{31} (-214,748.3648) through $2^{31} - 1$ (214.748.3647), with accuracy to ten-thousandth of a monetary unit.
`float`	floating precision number data type, ranging from -1.79 E+308 through 1.79 E+308.
`real`	floating precision number data type, ranging from -3.40 E+38 through 3.40 E+38.
`datetime`	date and time data type, ranging from January 1, 1753 through December 31, 9999, with accuracy to the millisecond.
`smalldatetime`	short date and time data type, ranging from January 1, 1900 through June 6, 2079, with accuracy to the minute.
`binary`	fixed-length binary data with a maximum length of 8,000 bytes. The length must be specified in the generic syntax `binary(n)`, where `n` is the length.
`varbinary`	variable-length binary data with a maximum length of 8,000 bytes. The length must

	be specified in the generic syntax `varbinary(n)`, where `n` is the maximum length.
`timestamp`	a database-wide unique number, this data type is semantically equivalent to `binary(8)`.
`char`	fixed-length non-Unicode character data with a maximum length of 8,000 bytes. The length must be specified in the generic syntax `char(n)`, where `n` is the length.
`varchar`	variable-length non-Unicode character data with a maximum length of 8,000 bytes. The length must be specified in the generic syntax `varchar(n)`, where `n` is the maximum length.
`nchar`	fixed-length Unicode character data with a maximum length of 4,000 bytes. The length must be specified in the generic syntax `nchar(n)`, where `n` is the length.
`nvarchar`	variable-length Unicode character data with a maximum length of 4,000 bytes. The length must be specified in the generic syntax `nvarchar(n)`, where `n` is the maximum length.
`Uniqueidentifier`	a globally unique identifier, to be used for referencing and manipulation of data similar to the *UNIQUEIDENTIFIER* globally unique identifier data type of MS-SQL Server.

In the rest of this section, we will outline more details about the `numeric`, `float`, `real`, `datetime` and `smalldatetime` data types, illustrated with some simple examples. We begin with the `numeric` data type. In this case, valid data values must be in the range from $-10^{28} + 1$ through $10^{28} - 1$, and will be stored with a maximum precision of 15 digits. Any number exceeding this limit will result to storage overflow run-time error. The following table illustrates acceptable valid values together with the corresponding stored values for a variable of `numeric` type:

Assigned value	Validation	Stored value
987654321098765.01234567890123456789	Valid Entry	987654321098765
987654321098765	Valid Entry	987654321098765
98765432109876543210987654321	Valid Entry	98765432109876500000000000000
987654321098765432109876543210	Overflow	(NULL)

Note that in the current version of the SCRIPT language, unlike the MS-SQL Server, the `numeric` data type has no precision and scale options. So that, this data type is suitable only for the purpose of manipulation and storage of decimal numbers comprising of at most 15 digits. Exceeding 15 digits will result in truncation and rounding.
We consider next more details for the `float` and `real` data types. The `float` type has the more general syntax `float(n)` where *n* is the number of bits used to store the mantissa of the floating point number, and therefore dictates the precision and storage size. In particular, if *n* is specified, it must be a value from 1 through 53. For *n*, with 0 <= *n* <= 24, the storage will be 4 bytes with precision to 7 digits. For *n*, with 25 <= *n* <= 53, the storage will be 8 bytes with precision to 15 digits. If *n* is not specified, by default the storage will be 8 bytes with precision to 15 digits. The `real` data type is semantically equivalent to `float(24)` type.
Note that, when assigning a value to a `float` variable, in non-scientific notation, the value must have the sum of the digits before and after the decimal point not exceeding 38. Otherwise, the value is out of range. Furthermore, valid values are approximated and stored with a precision of 7 digits or 15 digits, depending on the specified mantissa.
Finally, we close this subsection with some details about the `datetime` and `smalldatetime` data types. As noted above, a `datetime` type variable will take valid date-time values ranging from January 1, 1753 through December 31, 9999, with an accuracy to the millisecond. In contrast, `smalldatetime` data type variables can take valid date-time values ranging from January 1, 1900, through June 6, 2079, with accuracy to the minute. The following examples illustrate the rounding to the nearest minute:

Assigned value	Validation	Stored value
01/13/06 23:10:29.999	Valid Entry	2006-01-13 23:10
01/13/06 23:10:29.001	Valid Entry	2006-01-13 23:10

Declaration and Evaluation

Like all programming languages, SCRIPT language comes with formal variable declaration syntax to define the variable type, size, as well as, other properties. This variable declaration syntax along with the assignment syntax is very similar to the MS-SQL Server procedural language, except that in the SCRIPT language all statements must be terminated with the "`;`" terminator. In other words, unlike MS-SQL Server, the colon at the end of any

statement is not optional, and that the absence of this terminator will indeed generate a syntax error.
The generic syntax of the variable declaration is as follows:

```
DECLARE @variable_name data_type;
```

where *variable_name* is a valid variable name and *data_type* is any valid data type from the list of all types described in previous section. Observe the variable indicator prefix "@", which must prefix all basic variable names, and that the ";" terminator is required for each logical statement. Also, note that the SCRIPT language has a case insensitive syntax, so you may use your choice of upper or lower case conventions for coding.
A valid variable name comprises of a string of characters satisfying the following conditions:

(a) The string must start with any of the following characters: a ... z, A ... Z, or the underscore character "_".
(b) The rest of the string may be a sequence of alphanumeric characters. That is, any combination of characters in the following ranges: a ... z, A ... Z, 0 ... 9, or the underscore characters "_".
(c) And finally, variable names must not include any spaces.

We illustrate this with some examples (observe the usage of inline comments):

```
DECLARE @Last Name    varchar(50);
DECLARE @SSNum        char(9);       // only digits in social security number
DECLARE @IsInsured    bit;           // 1 for insured, 0 otherwise
DECLARE @Age          tinyint;
DECLARE @DivCode      smallint;      // division code is numeric value
DECLARE @EmployeeID   int;
DECLARE @AverScore    numeric;
DECLARE @DOB          smalldatetime; // date of birth
DECLARE @Income0_1    money;
DECLARE @Income0 2    money;
```

The scope of a declared variable extends throughout the entire SCRIPT file, regardless of the particular report section where it was originally declared. Moreover, all variables must be explicitly declared prior to their use.

The SCRIPT language deviates a little further from SQL procedural languages by the fact that it supports variable arrays. This makes the language a little more flexible, especially when a multitude of variables of the same type need to be defined. The generic syntax of declaring an array of a certain variable is as follows:

```
DECLARE @variable_name(n) data_type;
```

where *n* is any positive integer representing the number of array elements to be defined. Note that array indexing in SCRIPT language is zero-based. The following are some examples of declaration of arrays of variables:

```
DECLARE @Names(10)    varchar(50);
DECLARE @SSNums(10)   char(9);
DECLARE @Ages(10)     tinyint;
```

Observe that each of the array declaration in the example defines 10 indexed variable names, for example: @Names(0), @Names(1), @Names(2), @Names(3), @Names(4), @Names(5), @Names(6), @Names(7), @Names(8), @Names(9) for the first declaration, similarly for the other declarations.

We next describe the SCRIPT syntax for evaluation or assignment of an already declared variable. Essentially, variable assignment is performed via the SET keyword, with the following generic syntax:

```
SET @variable_name = value;
```

where *variable_name* is the declared variable's name, and *value* is any valid data value, validated with respect to the data type of the variable. The following SCRIPT code sample illustrates the declaration and assignment syntax for variables as well as variable arrays:

```
DECLARE @Last_Name          varchar(20);
DECLARE @First_Name         varchar(20);
DECLARE @SSNum              char(9);
DECLARE @MonthlyBonus(3)    money;
DECLARE @DOB                smalldatetime;
DECLARE @AFGScore           int;

SET @Last_Name = "EDISON";
SET @First Name = "THOMAS";
SET @SSNum = "457678976";
SET @DOB = "9/11/1945";     // SCRIPT engine validates and interprets date-time properly
SET @AFGScore = 89;
SET @MonthlyBonus(0) = 230.59;
SET @MonthlyBonus(1) = 301.34;
SET @MonthlyBonus(2) = 450.44;
```

Observe the evaluation of the array variable `@MonthlyBonus`, especially the 0-based nature of the array index. In particular, the array size is specified in the declaration, while the last array item's index is one less than the declared array size.
It is important to note that a date-time data type variable is often evaluated via a string constant which represents the desired valid date-time value (see the evaluation of `@DOB` variable). Internally, the SCRIPT engine will automatically convert the constant string, representing date-time value, into a date-time value prior to processing.
Finally, we should emphasize that, in SCRIPT language, there are three alternative methods of assigning long constant strings to a character data type variable. These methods are illustrated in the following sample code segments:

```
// this first method concatenates the value with new strings to produce the long string
DECLARE @MyString varchar(2000);

SET @MyString = "The story of SCRIPT is an interesting one. ";
SET @MyString = @MyString + "It all started back in 1998 when we were commissioned to ";
SET @MyString = @MyString + "develop a text-based print engine for one of our clients.";
```

An alternative method to produce essentially the same logical output as this first method is as follows:

```
// this method concatenates the value with a single SET keyword usage (or statement)
DECLARE @MyString varchar(2000);

SET @MyString = "The story of SCRIPT is an interesting one. " +
                "It all started back in 1998 when we were commissioned to " +
                "develop a text-based print engine for one of our clients.";
```

A third alternative method is as follows:

```
// this third method illustrates the multi-line capability of constant strings
DECLARE @MyString varchar(2000);

SET @MyString = "The story of SCRIPT is an interesting one.
                 It all started back in 1998 when we were commissioned to
                 develop a text-based print engine for one of our clients.";
```

Observe that in this latest method, double-quote specification extends beyond a single line. The SCRIPT engine will interpret these various assignment methods without any ambiguity.

Operations on Data Types

In this section we will outline the details concerning the operators used in the SCRIPT language. In particular, the SCRIPT language has numeric operations, string operations, comparison operations, logical operations, bit operations, and the assignment operation.

The numeric operations consist of the following: `+`, `-`, `*`, `/`, `%`, respectively *addition*, *subtraction*, *multiplication*, *division* and *modulo* operations. The *addition*, *subtraction*, *multiplication* and *division* operations are defined on any numeric data type, including the `bit`, `tinyint`, `smallint`, `int`, `real`, `float`, `numeric`, `smallmoney`, `money`, `smalldatetime` and `datetime`, while the *modulo* operation is applicable only to `tinyint`, `smallint` and `int` data types.
Because of the fact that there are several numeric data types, and that these types of variables are often mixed together in expressions built via numeric operations, it is necessary to address a few first order issues, such as:

- Precedence of operators in a mixed numeric expression
- Implicit conversion of data type in a mixed numeric expression before storing it in another variable
- Grouping subsections of a mixed numeric expression via parentheses

The precedence of numeric operators is as follows: first *division*, *multiplication* and *modulo* are applied and then *addition* and *subtraction*. For example in the expression of the following code snippet,

```
DECLARE @a, @b, @c, @d int;
DECLARE @X, @Y, @Z, @W int;

SET @X = @a*@b + @c/@d;       // do multiplication and division, and then the rest
SET @X = @a/@b*@c;            // do division first and then multiplication
SET @Y = (@a*@b) + (@c/@d);
SET @Z = @a*(@b + @c)/@d;     // force the addition and then do the rest
SET @W = -@a*+@b + @c;        // first the unary minus and plus and then the rest
```

first, multiplication of `@a` and `@b`, as well as, division of `@c` and `@d` are performed, and then the addition of the resulting sub-expressions is performed assigning it to `@X`. In essence, this is equivalent to the expression assigned to the variable `@Y`, which distinguishes from the previous expression only by proper grouping of sub-expressions via parentheses. Note also, that in the expression that is assigned to the variable `@Z`, the addition `@b + @c` is forced by the parenthesis to be performed prior to the multiplication and division operations.
There are also the *unary plus* and *unary minus* operations applicable to all numeric data types, essentially changing the sign value of an expression. For example, the last expression in the code snippet contains the sub-expression `-@a*+@b`, which has two unary operations. The precedence of operators in this case is as follows: first *unary plus* or *unary minus* operation must be performed, then *multiplication* or *division*, and then *addition* or *subtraction*. Therefore, the last expression should be equivalent to the following expression `@c - @a*@b`.

The next issue that we will consider is the implicit conversion of mixed types used in a numeric expression. When different numeric data types are used in a single expression, the SCRIPT compiler will automatically convert all data types to the largest data type used in the expression, so that no numerical precision error occurs during the evaluation of the expression. In the following example, prior to multiplication of `@a` and `@b`, variable `@b` is implicitly converted to `float` type. This is because `float` types have larger storage space than `int` data types. Similarly, before the addition is performed variable `@c` is also converted to `float` type.

```
DECLARE @a float;
DECLARE @b, @c, @X int;

SET @a = 37.89475;
SET @b = 11;
SET @c = 73;
SET @X = @a*@b + @c;  // the value on the right side becomes 489.84225
                      // but after assignment @X becomes 490
```

The example illustrates more than the implicit conversion mechanisms of the SCRIPT compiler. In particular, it also demonstrates the implicit rounding mechanism of the SCRIPT compiler. Getting back to the example, observe that the variable `@x` has type `int`, and it is being assigned to an expression which implicitly got converted to `float` type. Clearly, the value of the expression on the right must be implicitly rounded to integer value before the left side is assigned. This is precisely what happens when this code snippet is compiled and run by the SCRIPT engine. So that, the `float` data type value 489.84225 is rounded to 490 when it is stored into `@x`.
For those of you who were a little unfortunate in college years and never had the chance to follow advance courses in mathematics, the *modulo* operator is simply another way of expressing the remainder of integer divisions. So, for example 57 `%` 7 = 1, which means 57 divided by 7 gives you a remainder of 1. This operator is quite handy when there are periodic conditionals to be handled in some cases. For example, when checking parity of pages of a long report, or following some calendar event that repeats in certain frequencies, as well as, many other such similar situations.

We consider next the string operations. The only string operation is the string concatenation via the "`+`" operator. Moreover, constant string literals are defined by delimiting text in between double-quote. The following sample code illustrates all you need to know about string operations. Observe the various methods of assignment of long strings, which were already described earlier in previous sections.

```
DECLARE @sStory       varchar(500);
DECLARE @sBy          varchar(50);
DECLAER @sResult      varchar(2000);

SET @sStory = "The story of SCRIPT is an interesting one.
               It all started back in 1998 when we were commissioned to
               develop a text-based print engine for one of our clients.";

SET @sBy = " - SCRIPT's Architect";
SET @sResult = @sStory + @sBy;
```

We should emphasize that the concatenation operator applies particularly over string data types, these types are `char`, `varchar`, `nchar` and `nvarchar`. If a non-string data type is concatenated with string type, then a run-time type-mismatch error will occur when the report is run. We should also remark that the concatenation operator applies to `binary` and `varbinary` types, with a binary interpretation of the specified string value. The following table describes the possible combinations of the concatenation operations:

Combination	Validation	Storage
string_type + *string_type*	Valid entry	The concatenation of strings
string_type + *numeric_type*	Type-mismatch error at run-time	No effect
string_type + *datetime_type*	Type-mismatch error at run-time	No effect
string_type + *unicode_string_type*	Valid entry	The concatenation of strings in Unicode
string_type + *binary_type*	Valid entry	The concatenation in binary type
binary_type + *binary_type*	Valid entry	The concatenation in binary type
unicode_string_type + *unicode_string_type*	Valid entry	The concatenation of strings in Unicode

Now we dive into the subject of comparison operations in the SCRIPT language. Well, pretty much we have all possible comparison operators, in fact, a little redundant too. These operators are: `=`, `>`, `<`, `>=`, `<=`, `<>`, `!=`, `!>`, `!<`, respectively *equal*, *greater*, *less*, *greater or equal*, *less or equal*, *not equal* (alternate notation), *not equal*, *not greater*, and *not less*. Note that there are two alternative notations for *not equal* operation. Also, we emphasize that the equality comparison operator and the assignment operator cannot be confused in the SCRIPT language, since an assignment equality operator is distinguished from the equality comparison operator by the fact that the `SET` keyword precedes the statement of assignment.
Before dwelling into some examples regarding the comparison operations, it is crucial to also consider the logical operations. The SCRIPT language has a complete set of logical operators consisting of the `NOT` unary operator, the `AND` and `OR` binary operators, with the stated order of precedence. Here are some examples of usage of the comparison, logical, as well as, other operators:

```
DECLARE @OddLine, @IsNegative BIT;          // declaring multiple variables
DECLARE @Category VARCHAR(20);
```

```
SET @Category = "UNKNOWN";
IF Amount < 0 AND @OddLine != 0 THEN
     SET @Category = "BAD CREDIT";
     SET @IsNegative = 1;
ELSE IF @OddLine = 0 OR Amount > 100
                AND NOT @IsNegative THEN    // breaking line is OK
     SET @Category = "GOOD CREDIT";
     SET @IsNegative = 0;
ELSE
     SET @Category = "GIVE CREDIT";
END IF
SET @OddLine = NOT @OddLine;                // applying unary negation to toggle value
```

Note that in the example code above we have used the conditional controls `IF`, `ELSE IF`, and `ELSE`. The syntax of the conditional and iteration controls will be described in the next subsection.

Finally we consider the bit operations, essentially consisting of `~`, `&`, `|`, `^`, respectively the *bitwise unary negation* operator, the *bitwise and*, *or* and *xor* (i.e. exclusive or) binary operators, with the stated precedence order. These operators are applicable only on the integer data types, including `bit`, `tinyint`, `smallint` and `int`, with respectively different results depending on variable's declared type. The definitions of these operators over the `bit` data type are defined as follows:

```
~@b0 = 1
~@b1 = 0
```

for *bitwise unary negation* operator,

```
@b0 & @b1 = 0
@b1 & @b1 = 1
@b1 & @b0 = 0
@b0 & @b0 = 0
```

for *bitwise and* operator,

```
@b0 | @b1 = 1
@b1 | @b1 = 1
@b1 | @b0 = 1
@b0 | @b0 = 0
```

for *bitwise or* operator, and

```
@b0 ^ @b1 = 1
@b1 ^ @b1 = 0
@b1 ^ @b0 = 1
@b0 ^ @b0 = 0
```

for *bitwise xor* operator, where `@b0` and `@b1` are variables of `bit` data type, set to value 0 and `@b1` set to value 1. The following code sample illustrates the use of the bit operations:

```
DECLARE @b0, @b1, @W BIT;

SET @b0 = 0;
SET @b1 = 1;
SET @W = ~@b0 & @b1 | @b0 ^ ~@b1;     // first apply the unary bitwise NEGATION operator,
                                      // then the bitwise AND operator on ~@b0 and @b1,
                                      // then the bitwise OR operator with @b0,
                                      // finally apply bitwise XOR operator with ~@b1,
                                      // the result becomes 1
```

We extend now the definitions of bit operations respectively over `tinyint`, `smallint` and `int` data types.

Basically, the operation is similar to the `bit` data type case, but applied over the bits of the value represented in binary form. For example, in `tinyint`, the number `7` and `5` have binary representations `0000 0111` and `0000 0101`, so that the `~`, `&`, `|` and `^` operations yield the following:

`~7` is computed by negating bitwise all the bits, which yields: `1111 1000`, which is equal to `248`,
`~5` is computed by negating bitwise all the bits, which yields: `1111 1010`, which is equal to `250`,
`~7 & 5`, `~5 | 7` and `5 ^ 7` are computed by applying the bitwise operations over all the respective bits, which yield respectively the values `0`, `255` and `2`:

```
  1111 1000          1111 1010          0000 0101
& 0000 0101        | 0000 0111        ^ 0000 0111
  ---------          ---------          ---------
  0000 0000          1111 1111          0000 0010
```

In a very similar fashion the bit operations may be bitwise computed over 16-bits for `smallint`, and 32-bits for `int` data types. For example, the values of the expressions `~7 & 5`, `~5 | 7` and `5 ^ 7` in both integer types are computed to yield respectively `0`, `-1` and `2`.

Conditional and Iteration Controls

In the business world, even the simplest reports follow some sort of logical structure in which condition checking and iteration are indispensable elements. In fact, any useful, down to earth report often contains sections consisting of running totals, summaries, categories, groups and other such constructs. Sometimes such summaries are so standard and hierarchical that standardized functions applied via graphic interfaces are extremely suitable for accomplishing such tasks. In some other cases however, the summary or grouping logic is so arbitrary that a more structural scripting approach becomes inevitably desirable and much more suitable. It is to this later cause that SCRIPT language supports full conditional and iteration controls, essentially very similar to Visual Basic or MS-SQL Server procedural languages.

The simplest version of the conditional control in SCRIPT language has the following `IF-THEN` form:

```
IF statement THEN
        result_statement_1;
        result_statement_2;
        ...
        result_statement_N;
END IF
```

where *result_statement_1*, *result_statement_2*, ... , *result_statement_N*, form a sequence of resulting actions. The most generic form of this conditional control contains optional one or more `ELSE IF` branches and an optional `ELSE` branch, as is shown in the following generic example:

```
IF statement_1 THEN
        result_statement_1;
        result_statement_2;
ELSE IF statement_2 THEN
        result_statement_3;
ELSE IF statement_2 THEN
        result_statement_4;
        result_statement_5;
ELSE
        result_statement_6;
        result_statement_7;
END IF
```

Recall that we have already illustrated the use of this more general form of the conditional control in the code example for the comparison operators, outlined earlier in this section.
In addition to the `IF-ELSE IF-ELSE` conditional, SCRIPT language has an expression level `CASE` conditional very similar to the MS-SQL Server's *simple case* syntax. The syntax of simple `CASE-WHEN-ELSE` expression conditional is as follows:

```
CASE in_expression
        WHEN in_expression_value_1 THEN out_expression_1
        WHEN in_expression_value_2 THEN out_expression_2
        ELSE out_expression_3
```

```
END CASE;
```

Where *in_expression* is the input expression, *in_expression_value_1* and *in_expression_value_2* are input expression's possible values, *out_expression_1*, *out_expression_2* and *out_expression_3* are respective output expressions. We should emphasize that at least one WHEN-phrase is required in the syntax of the simple CASE, and that the ELSE-phrase is optional and may be omitted if not needed. Also, observe that this conditional is defined on expressions and not on logical statements. In fact, the whole CASE construct is itself a single statement, as is apparent from the single semi-colon at the end of the END CASE keyword. We illustrate next the usage of this simple CASE conditional in the following:

```
DECLARE @nCode INT;
DECLARE @vCode VARCHAR(10);

SET @vCode = "CABINET";     // this line is just for test

// a translation from character codes to integer values
SET @nCode = CASE @vCode
                WHEN "CABINET" THEN 1100
                WHEN "CLOSET" THEN 1103
                WHEN "GARAGEDOOR" THEN 2145
                WHEN "OFFICEDESK" THEN 3452
                ELSE 1000
          END CASE;
// at this point the value of @nCode becomes 1100
```

Finally, in SCRIPT language the iteration control is defined by the WHILE, END LOOP, CONTINUE and BREAK keywords. The general syntax is as follows:

```
WHILE check_statement
       statement_1;
       statement_2;
       ...
       statement_N;
END LOOP
```

where *check_statement* is the statement on which the iteration is defined, and *statement_1*, *statement_2*, ... , *statement_N* are statements applied per iteration. Moreover, the BREAK keyword is used to conditionally exit the loop from inside, and the CONTINUE keyword is used to jump to the next iteration from the current location inside the loop. The following code snippet illustrates the use of the iteration control:

```
// The following code generates the content of a calendar month
DECLARE @i, @nDay     INT;
DECLARE @nOffset      SMALLINT;
DECLARE @vA(42)       VARCHAR(200);
DECLARE @dFD, @dTmp   DATETIME;

// get the correct date of the first day of the current month
SET @dFD = ToDate(Format(GetDate(), "yyyy-mm-01"), "yyyy-mm-dd");
// find the day of the week of the date @dFD
SET @nOffset = CAST(Format(@dFD, "w") AS INT);

// loop over the days and fill the array @vA
WHILE @i < 50
     // initialize value to the HTML space character
     SET @vA(@i) = " ";
     // all cells before the 1st day should be filled with space
     IF @i >= @nOffset-1 THEN
          // add @nDay to @dFD date
          SET @dTmp = DateAdd(Day, @nDay, @dFD);
          // if new value is in the current month get the day
          IF DatePart(MM, @dTmp) = DatePart(MM, @dFD) THEN
               SET @vA(@i) = Format(@dTmp, "d");
```

```
            END IF
            // for the current day, highlight cell with yellow background
            IF Format(@dTmp, "yyyy-mm-dd") = Format(GetDate(), "yyyy-mm-dd") THEN
                  SET @vA(@i) = "<SPAN style='background-color:#FFFF00'" +
                                    @vA(@i) + "</SPAN>";
            END IF
            SET @nDay = @nDay + 1;
      END IF
      SET @i = @i + 1;
      IF @i = 42 THEN
            // exit loop if we get to 42 days (i.e. 7x6 month calendar)
            BREAK;
      END IF
END LOOP
```

Observe that the above code sample included several functions that we have not yet explored, namely: `ToDate`, `Format`, `GetDate`, `Cast`, `DateAdd`, and `DatePart`. We will define the details of these functions later in this chapter. Also, observe that the iteration is actually creating HTML segments containing the day's number, and for the current day it highlights the cell yellow via the `<SPAN>` tags. It is not hard to see that this code sample may be used to generate nice looking calendars in HTML format. We will get back to this sample later in the "Functions and Macros" subsection.

Datasource Object and FLD-tag

A data oriented report tool without a data source support is not worth using. The SCRIPT language comes with built-in data connection, data source, data shape and XML objects, for retrieval of data from various data stores. These objects behave more like data objects contrast to the basic data types. In particular a variable may be declared of such data object type, and inherit all properties and methods of these object types.
In this section we will only describe the `DATASOURCE` object, deferring the rest to later sections in this chapter. Variable declaration for the `DATASOURCE` object is as follows:

```
DECLARE @variable_name DATASOURCE;
```

where *variable_name* is the name of the variable of data source type. Such objects will inherit all properties and methods of the internal data source object, which we list next:

`Connect(CONNECTION objConn)`	function to define the data connection for the data source, if this function is never called for a particular `DATASOURCE` object, then the default connection of the report engine will be assumed. The argument must be an object of `CONNECTION` type (which will be described later in this chapter).
`Column(VARCHAR FieldName)`	function to return the value of a particular field of the data source, returns a data type depending on the data type of the field in the backend database server. The argument must be a `VARCHAR` string representing a valid name of the field or alias of the field.
`Child(VARCHAR FieldName)`	function to return the data shape object defined by a field in the original data shape command with respect to the backend database server. The argument must be the name of a valid field or alias name in the data shape command.
`EOF()`	function to return `bit` value representing end of file mark of the recordset defined by the data source.
`Next()`	function to advance the recordset cursor one record ahead.
`Reset()`	function to reset the recordset cursor to empty and reopen the recordset defined by the latest `SET` call for the data source object.

The following code snippet describes the usage of the data source object:

```
<html>
<body>
```

```
// put report settings and other elements here...

// set the font to Arial 8pt
<font style="font-family: arial; font-size:8pt">
<ELS>      // contain pure SCRIPT in <ELS>-tags
DECLARE @ds           DATASOURCE;
DECLARE @nRecCount    INT;

// set or define the data source
SET @ds = "SELECT OrderID, OrderDate, ShipVia, ShipName AS SName, " +
          "ShipCity AS SCity, ShipCountry AS SCountry " +
          "FROM Orders " +
          "WHERE ShipVia > 2";

// iterate over the recordset, note that the cursor opens automatically
WHILE NOT @ds.EOF()
</ELS>     // switch the <ELS> tag off to specify HTML code or fields
     <b>Order Date: </b><FLD>@ds.Column("OrderDate")</FLD><br>
     <b>Ship Name: </b><FLD>@ds.Column("SName")</FLD><br>
     <b>Ship City: </b><FLD>@ds.Column("SCity")</FLD><br>
     <b>Ship Country: </b><FLD>@ds.Column("SCountry")</FLD><br><br>
<ELS>      // get back into <ELS>-tag to specify the rest of the loop
     // advance by one record
     @ds.Next();
     // increment the record count
     SET @nRecCount = @nRecCount + 1;
END LOOP
</ELS>     // get out of SCRIPT code into the HTML code
<hr>
<b>Total Number of Records: </b><FLD>@nRecCount</FLD>
</font>
</body>
</html>
```

We list a few records from the resulting output to this code sample:

Order Date: 1996/07/04 00:00:00.000
Ship Name: Vins et alcools Chevalier
Ship City: Reims
Ship Country: France

Order Date: 1996/07/12 00:00:00.000
Ship Name: Richter Supermarkt
Ship City: Genève
Ship Country: Switzerland

Order Date: 1996/07/16 00:00:00.000
Ship Name: HILARION-Abastos
Ship City: San Cristóbal
Ship Country: Venezuela

Since this sample code is a little involved, we find it necessary to give a more detailed exposition, essentially stepping through each line of the sample code. The first basic observation is the fact that the SCRIPT language, just like any web script language, may be mixed with HTML code. In particular, part of this sample illustrates how pure SCRIPT code may be mixed with HTML in a natural nested fashion, with the whole report code contained in `<HTML>` tags, resembling an HTML document.
We begin with the `<font>` tags that wrap around the SCRIPT code, in effect, defining the typeface for the resulting output. The next line of the code opens an `<ELS>` tag to switch to pure SCRIPT code. The data source variable `@ds` is declared as a `DATASOURCE` object, along with other variable declarations. The data source variable `@ds` is then set to the desired SQL statement, which may have alias names, and could be any valid query statement based on the backend database server's language. Note that the SQL statement is included as a character string, and that the double quotes must surround it, just like any character value specification. This will inform the SCRIPT engine to

treat this SQL statement as a constant character string value (and therefore, skip compiling this segment).
Then we loop over the recordset defined by this data source. In particular, observe how the `EOF()` and `Next()` methods are used. To output text we switch mode to HTML via the `</ELS>` tag, in which HTML code segment or recordset field values are put via the `<FLD>` tags of SCRIPT language. In particular, observe how the `Column()` function is used with arguments being field names or alias names of fields. The HTML `<b>` tag makes the field labels in bold font style, while `<br>` is the HTML to break the line of text. The mode switches back to pure SCRIPT code via the `<ELS>` tag, followed by the `Next()` function call to advance the recordset cursor by one record. Then the record count is incremented and the loop continues. At the end of all records, the mode switches back to HTML, and the HTML `<hr>` is put to add a horizontal line in the output. Then the total number of records is put via the `<FLD>` tags.
In the sample code above we have already utilized the `<FLD>` tag of the SCRIPT language, which we will need to explain next. In particular, the `<FLD>` tag is used to put the result of any expression or field into the report output as HTML segment. It can be used only outside pure *ELS* sections, that is, outside the `<ELS>`, `<ELS_QPARAMS>`, `<ELS_RSETTINGS>` and `<ELS_OPROCS>` tag sections. Getting back to the code sample, observe for example, that the `</ELS>` tag switches the scope of pure ELS section off, and that it is outside this `<ELS>` tag section where the `<FLD>` tags are used.
The content of the `<FLD>` tag must be pure *ELS* elements, which may consist of valid SCRIPT expressions or data fields. Moreover, the main function of the `<FLD>` tag is to convert the run-time value of any valid SCRIPT expression to the character string representation, to be put as HTML segment into the report output.

The SCRIPT Editor

To boldly go where no human has gone before, you will need the SCRIPT *Editor*, which comes with the SCRIPT compiler. In this section we will describe most of the important details concerning the utilization of the SCRIPT *Editor* module included in the *Report Designer* application. In particular, we will begin with an exposition of the GUI features, including the menu commands, toolbars, various satellite dialogs and windows. We will then describe the following functions or edit operations:

- Creating a new report via the standard report templates, and maintaining it entirely in **Source** view of the editor
- Defining the report settings parameters via the **Report Settings** dialog and the **Source** view
- Defining data source and using the **Data Fields** and the **Expression Builder** to insert data fields and expressions into the **Source** view
- Performing basic edit operations, as well as, compiling and running the report

Using the Editor

The SCRIPT *Editor* is the ultimate place to be when flexibility and precision are the top priority issues in the making of a report. It is the sanctuary, the ultimate refuge of the script oriented developer, whose daily thoughts and ideas are full of nightmares concerning ASP, VB or Java scripting mixed with DHTML code. As we have already learned that SCRIPT language behaves very similar to mixed scripting languages, conforming its content or functionality via tags, just like other scripting languages, over the domain of HTML code. More concisely, they regenerate and manipulate the elements of the underlying HTML document structure in possibly almost unlimited combinations. The only difference from the other scripting languages is the fact that the SCRIPT language code is intended to be deployed at run-time in a compiled binary form. In this way, no time is lost in parsing the syntax of SCRIPT segments when the report output generation is triggered. Moreover, part of main tasks of the compilation process is to optimize SCRIPT instructions, as well as, HTML elements into a binary structure, in such a way, so that the report generation process becomes extremely fast and least resource consuming.

We will begin the description of the features of the SCRIPT *Editor* by outlining the menu commands together with the corresponding toolbars. Recall that the menu bar of the *Report Designer* application consists of the **File**, **Edit**, **View**, **SQL**, **Insert**, **Format**, **Position**, **Table**, **Build**, **Tools**, **Windows** and **Help** main menus. The **File** menu contains the following menu items:

New Project...	**Ctrl+N**	this will call the **New Project** dialog,
Open Project...	**Ctrl+O**	this will call the **Open Project** dialog,

Close Project		this will close the currently open project,
New Report...		this will call the **New Report** window,
Open Report...		this will call the **Open ELS Report File** dialog,
New SQL Script		this will create a new instance of SQL script window into the *SQL Editor*,
Open SQL Script...		this will call the **Open Query File** dialog,
New DB Connection...		this will call the **New Database Connection** dialog,
Add To Project...		this will call the **Add to Project** dialog,
Remove From Project		this will remove the selected file from the **Project Explorer** view,
SQL Dictionary...	**F12**	this will call the **SQL Dictionary** window,
Save	**Ctrl+S**	this will save the changes made to the currently open report script,
Save All		this will save changes made to all open report files,
Save As ...		this will call the **Save ELS Script** dialog,
Save HTML ...		this will call the **Save HTML Output** dialog,
Print ...	**Ctrl+P**	this will call the **Print** dialog to print the content of **Source** view,
Print Current Page		this will print the report output page displayed in the report viewer,
Print Preview		this will display the content of **Source** view in a preview window,
Page Setup ...		this will call the **Page Setup** dialog to setup page for printing source,
Report Settings ...		this will call the **Report Settings** dialog,
Recent Files ►		keeps a list of recently opened report files,
Recent Project ►		keeps a list of recently opened projects,
Exit		exits the application.

The **Edit** menu contains the following menu items:

Undo	**Ctrl+Z**	this will undo last edit operation,
Redo	**Ctrl+Y**	this will redo the last undone operation,
Cut	**Ctrl+X**	this will cut the current selection and copy it into the clipboard,
Copy	**Ctrl+C**	this will copy the current selection into the clipboard,
Paste	**Ctrl+V**	this will paste the content of the clipboard into the cursor location,
Select All	**Ctrl+A**	this will select all text in the **Source** view,
Find...	**Ctrl+F**	this will call the **Find** dialog,
Find in Files...		this will call the **Find In Files** dialog,
Replace...	**Ctrl+H**	this will call the **Replace** dialog,
Toggle Bookmark	**Ctrl+F2**	this will toggle the bookmark for the current line,
Next Bookmark	**F2**	this will shift the view of **Source** view to the next bookmark,
Previous Bookmark	**Shift+F2**	this will shift the view of **Source** view to the previous bookmark,
Clear All Bookmarks	**Ctrl+Shift+F2**	this will clear all current bookmarks,
Clear All Breakpoints		this will clear all breakpoints in the active report script,
Advanced	►	this menu has two submenus to change the letter case of the selected.

The **Advanced** menu item has the following two child menus:

Make Selection Lower Case	**Ctrl+U**	this will make the selected text all in lower case,
Make Selection Upper Case	**Ctrl+Shift+U**	this will make the selected text all in upper case.

The **View** menu contains the following menu items:

Toolbars...		this will call the **Customize** dialog for toolbar setup,
Project Explorer	**Ctrl+E**	this will toggle the visibility of the **Project Explorer** pane,
Properties/Data View	**Ctrl+D**	this will toggle the visibility of the **Properties** pane,
Status Bar		this will toggle the visibility of the status bar line,
Full Screen		this will maximize the active view to full screen,
---		
Text Size	►	this consists of submenus to change the text size in the viewer,
---		
Options...		this will call the **Options** dialog.

The **SQL** menu is destined for SQL query operations, and therefore many of the items of this menu are not applicable when the **Source** view is active. In fact, the only applicable menu items are the following:

New Query ►	this consists of submenus to create various kinds of queries,
New Data Shape...	this will slide on the **Data Shape** sliding window,
---	
... *other menu items* ...	

SQL Dictionary...	this will call the **SQL Dictionary** window,

where the **New Query** menu has the following submenus:

SELECT...	this will open an instance of query window in *SELECT* command mode,
INSERT...	this will call the **Insert Into Table** dialog to start an *INSERT* query,
INSERT VALUES...	this will open an instance of query window in *INSERT VALUES* command mode,
UPDATE...	this will open an instance of query window in *UPDATE* command mode,
DELETE...	this will open an instance of query window in *DELETE* command mode,
Make Table...	this will call the **Make Table** dialog to start a *CREATE* query mode.

The **Insert** menu contains the following menu items:

Break		this will insert a ` ` HTML tag at the cursor location,
Rule		this will insert a `<HR>` HTML tag at the cursor location,
New Paragraph		this will insert an empty `<P></P>` HTML paragraph at the cursor location,
New Section		this will insert an empty `<DIV></DIV>` HTML section at the cursor location,
Append New Section	**Ctrl+W**	this will append an empty new section at the end of the current section,
---		
Special Symbols...		this will call the **Special Symbols** dialog,
---		
Picture...		this will call the **Select Image File** dialog,
Background Image...		this will call the **Select Background Image** dialog,
---		
HTML Table...		this will call the **Insert Table** dialog,
ELS Row...		this will call the **Insert ELS-Row** dialog
ELS Shape...		this will call the **Insert ELS-Shape** dialog,
Expression Builder...		this will call the **Expression Builder** window,
Data Fields...		this will call the **Data Fields** window,

ActiveX Control...		this will call the **Insert ActiveX Control** dialog,

We should emphasize that whereas the **New Section** menu item just inserts the string `<DIV></DIV>` at the current cursor location, the **Append New Section** menu item, on the other hand, inserts the HTML *DIV*-element with complete style attributes, and that this insertion is at the end of the currently selected report section. For example, if the current cursor's position is inside the *ELS_RDETAIL* section, then the insertion will be just before the `</ELS_RDETAIL>` end tag. In general, for sections *ELS_QPARAM*, *ELS_OPROCS*, *ELS_RSETTINGS* and *ELS_RDETAIL*, the **Append New Section** menu's insertion will be at the end of the *ELS_RDETAIL* section just before the `</ELS_RDETAIL>` end tag. For *ELS_RHEADER* section, this insertion will be at the end of the

ELS_RHEADER section just before the `</ELS_RHEADER>` end tag. For *ELS_RFOOTER* section, this insertion will be at the end of the *ELS_RFOOTER* section just before the `</ELS_RFOOTER>` end tag. Similarly, for *ELS_PHEADER* or *ELS_PFOOTER* sections, this insertion will respectively be at the end of the *ELS_PHEADER* or *ELS_PFOOTER* sections, and respectively, just before the `</ELS_PHEADER>` and `</ELS_PFOOTER>` end tags. Moreover, the string that is inserted by the **Append New Section** menu command will have the following form:

```
<DIV style=font_attributes></DIV>
```

where the *font_attributes* is the style attributes of the default typeface for the current report section obtained from the settings in the **Options** dialog. For example, if in the **Options** dialog the report detail section is set to have Trebuchet MS Bold 8pt typeface with blue text color, then if the user puts the cursor anywhere in the *ELS_RDETAIL* section, and triggers the **Append New Section** menu command the resulting appended string will be as follows:

```
<DIV style="font-family:Trebuchet MS; font-size:8pt;
                    font-style:bold; color:blue"></DIV>
```

The **Format** and **Position** menus have items that do not apply to the SCRIPT *Editor* module, while the only applicable menu item in the **Table** menu is the **Insert ELS Line** menu item. This menu command will call the **Insert ELS Line** dialog, with which one may insert a special kind of *ELS Line* element (to be described soon).
The **Build** menu contains the following menu items:

Compile		this will trigger the compilation of the current report script,
Cancel		this will stop any active compilation process,
Build		this will trigger the build of all the report scripts in the current project,
---		
Debug		this will activate the debugging process for the current report script,
Go	**F6**	this will run the report generation process till the next debug breakpoint,
Step Over	**F10**	this will process the next statement in the script,
---		
Run		this will run the report generation process,
Stop		this will stop any active report generation process,

First Page		this will display the first page of the report output,
Previous Page		this will display the previous page of the report output,
Next Page		this will display the next page of the report output,
Last Page		this will display the last page of the report output.

Finally, the **Tools** menu contains the following menu items:

Connection / Datasource List...	this will call the **Connection/Datasource List** window,
Parameter List...	this will slide on the **Parameter List** sliding window,
Query Form...	this will slide on the **Query Form** sliding window,
Precision Resizer...	this will call the **Precision Resizer** window,
Show Report Wizard	this will make an active *SRT Wizard* window visible,
---	
Edit SQL Dictionary...	this will call the **SQL Dictionary** in edit mode,
---	
Transform	this will be used to transform between SCRIPT and *RDL* reports.

Now, we outline the toolbars of the *Report Designer* application that are relevant to the SCRIPT *Editor*. In particular, the designer application has the following predefined toolbars: **Project Explorer**, **Compile**, **Navigate**, **HTML**, **Borders**, **SQL**, **Report Design**, **Position** and **Table** toolbars. Of these, only the **Project Explorer**, **Compile**, **Navigate** and **Report Design** toolbars are applicable to the SCRIPT *Editor*.

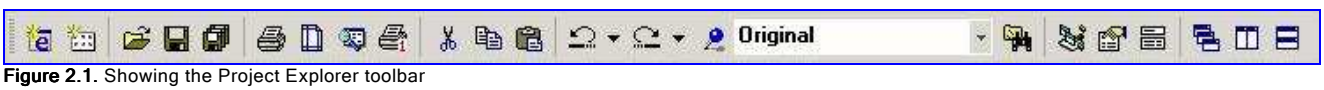


Figure 2.1. Showing the Project Explorer toolbar

The **Project Explorer** toolbar consists of the following command buttons (see Figure 2.1):

New Project	this will call the **New Project** dialog (same as the **New Project** menu),
Add Item	this will call the **Add to Project** dialog (same as the **Add to Project** menu),
- - - - - - - - - - - - - - -	
Open	this will call the **Open ELS Report File** dialog (same as the **Open Report** menu),
Save	this will save the changes made to current report script (same as the **Save** menu),
Save All	this will save the changes to all open report files (same as the **Save All** menu),
- - - - - - - - - - - - - - -	
Print	this will call the **Print** dialog (sane as the **Print** menu),
Page Setup	this will call the **Page Setup** dialog (same as the **Page Setup** menu),
Preview	this will preview the content of **Source** view (same as **Print Preview** menu),
Print Current Page	this will print the current output page (same as the **Print** menu),
- - - - - - - - - - - - - - -	
Cut	this will cut the selected text retaining a copy in the clipboard (same as **Cut** menu),
Copy	this will copy the selected text into the clipboard (same as **Copy** menu),
Paste	this will paste the content of the clipboard into **Source** view (same as **Paste** menu),
- - - - - - - - - - - - - - -	
Undo	this will undo the last edit action (same as **Undo** menu),
Redo	this will redo the last undo operation (same as **Redo** menu),
Toggle Bookmark	this will toggle the bookmark of the current line (same as **Toggle Bookmark** menu),
- - - - - - - - - - - - - - -	
Search	a combo-box used for search text entry, the user must click the *Enter* key to start search, subsequent searches of the same keyword may be performed via **F3** key,
Find In Files	this will call the **Find In Files** dialog (same as **Find In Files** menu),
- - - - - - - - - - - - - - -	
Project Explorer	this will open (or set focus on) the **Project Explorer** pane (same as **Project Explorer** menu),
Properties	this will open (or set focus on) the **Properties** pane (same as **Properties/Data View** menu),
Show/Hide Results Pane	this will show / hide the **Results** pane,
- - - - - - - - - - - - - - -	
Cascade Windows	this will cascade all open windows
Tile Windows Vertical	this will tile all open windows vertically,
Tile Windows Horizontal	this will tile all open windows horizontally,

Figure 2.2. Showing the Report Design toolbar

The **Report Design** toolbar consists of the following command buttons (see Figure 2.2):

Design View	this will display the current report in **Design** view,
Code View	this will display the current report in **Source** view,
- - - - - - - - - - - - - - -	
Open	this will open the currently selected report in the **Project Explorer** pane,
- - - - - - - - - - - - - - -	
Insert ELS Row	this will call the **Insert ELS-Row** window (same as **ELS Row** menu),
Insert ELS Shape	this will call the **Insert ELS-Shape** window (same as **ELS Shape** menu),
Insert HTML Table	this will call the **Insert Table** window (same as **HTML Table** menu),
Insert Image	this will call the **Select Image File** dialog (same as **Picture** menu),
Insert Expression	this will call the **Expression Builder** window (same as **Expression Builder** menu),
Insert Date Fields	this will call the **Data Fields** window (same as **Data Fields** menu).

Figure 2.3. Showing the Compile and Navigate toolbars

The **Compile** / **Navigate** toolbars consist of the following command buttons (see Figure 2.3):

Compile ELS-file	this will compile the current *ELS*-file (same as **Compile** menu),

Cancel	this will cancel any active compilation process (same as **Cancel** menu),
Run	this will trigger the report generation process (same as **Run** menu),
Stop	this will stop any active report generation process (same as **Stop** menu),
- - - - - - - - - - - - - - -	
First Page	this will display the first page of the report output (same as **First Page** menu),
Previous Page	this will display the previous page of the report output (same as **Previous Page** menu),
Move To	edit-control, to enter page number abd press the *Enter* key to jump to the page,
Move To Page	an edit-box to enter page number, the user must click *Enter* key to jump to that page,
Next Page	this will display the next page of the report output (same as **Next Page** menu),
Last Page	this will display the last page of the report output (same as **Last Page** menu).

We describe next two popup menus that are relevant to the SCRIPT *Editor*, namely, the **Source** view and the **Project Explorer** pane popup menus. The **Source** view popup menu consists of the following menu items:

Undo		this will undo the last edit action,
Redo		this will redo the last undo operation,

Cut		this will cut the selected text and copy it to clipboard,
Copy		this will copy the selected text into the clipboard,
Paste		this will paste the content of the clipboard into the **Source** view,
----------		
Select All		this will select the whole content of the **Source** view,
----------		
Select for Auto-Align		this will select the current line into the **Auto-Align** dialog,
----------		
Open Document		this will open the URL or path that is selected in the **Source** view,
Show IntelliSense	**F5**	this will show the *IntelliSense* corresponding to the selected context,
Parameter Info		this will show the parameter structure of the selected object in tool-tip,
List Report Settings		this will list the report settings parameters,
List CSS Properties		this will list the Cascaded Style Sheet properties,
----------		
Add/Move To Section	►	this menu consists of submenus for each report section,

where the submenus of the **Add/Move To Section** menu item are as follows:

ELS_RDETAIL	this will move the cursor position to the *ELS_RDETAIL* section,
ELS_PHEADER	this will move the cursor position to the *ELS_PHEADER* section,
ELS_PFOOTER	this will move the cursor position to the *ELS_PFOOTER* section,
ELS_RHEADER	this will move the cursor position to the *ELS_RHEADER* section,
ELS_RFOOTER	this will move the cursor position to the *ELS_RFOOTER* section,
----------	
ELS_RSETTINGS	this will move the cursor position to the *ELS_RSETTINGS* section,
ELS_QPARAMS	this will move the cursor position to the *ELS_QPARAMS* section,
----------	
OnBeginPage	this will move the cursor position to the *OnBeginPage* event handler section,
OnEndPage	this will move the cursor position to the *OnEndPage* event handler section,
OnBeginReport	this will move the cursor position to the *OnBeginReport* event handler section,
OnEndReport	this will move the cursor position to the *OnEndReport* event handler section.
OnSendMessage	this will move the cursor position to the *OnSendMessage* event handler section.

The **Project Explorer** pane's popup menu, relative to *ELS*-file objects in the tree-view, consists of the following submenus:

Add Item...	this will call the **Add to Project** dialog (same as **Add to Project** menu),
New...	this will call the **New Report** dialog (same as **New Report** menu),
----------	
Open	this will open the report selected in the **Project Explorer** pane,

Properties...	this will call the **Project Properties** dialog, when activated from the project node,
Rename	this will make the selected node's name editable, so that the user may change the name,
---	
Remove	this will remove the selected object from the project,
Copy	this will copy the object into the clipboard,
Paste	this will paste objects that may exist in the clipboard into the project,
---	
Allow Docking	this will toggle the mode of the pane docking or non-docking,
Hide	this will hide the **Project Explorer** pane.

The SCRIPT *Editor* comes with full *IntelliSense* support making the SCRIPT language extremely intuitive to use. This service helps the user by automatically popping up selection lists, with content depending on the context. Sometimes it is necessary to force the *IntelliSense* to show such lists on demand. This may be achieved via the popup menu or with the **F5** shortcut key. Similarly, to see the parameter information of a function entered in the **Source** view, one may use the **Parameter Info** menu item from the popup menu.

It is perhaps more suitable to describe the various windows and dialogs of the *Report Designer* along with a sample application, rather than just listing all features like a reference. We will use this approach in the next section titled "A Simple Example". But before we delve into that realm we should describe the role of the **Options** dialog (see Figure 2.4).

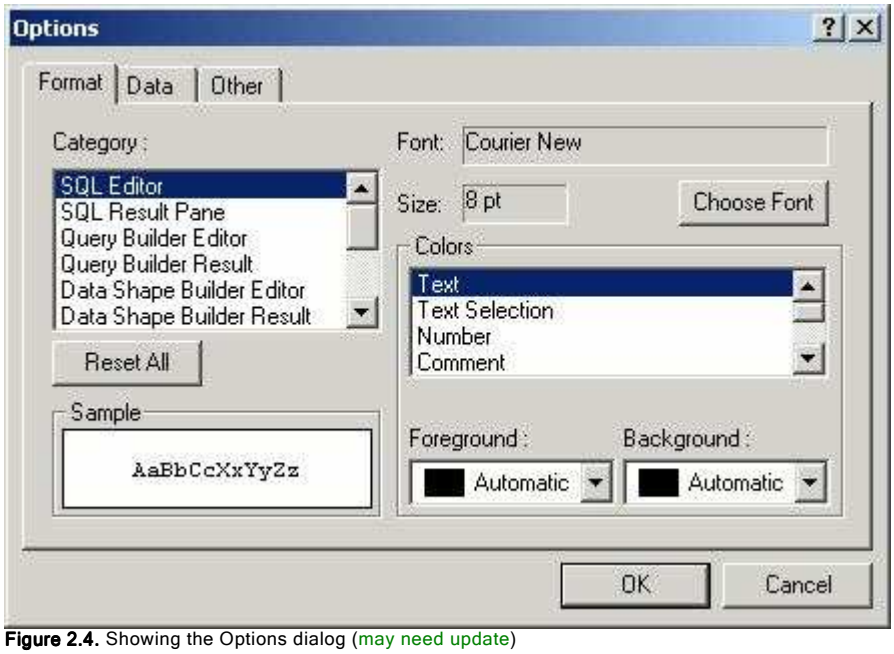


Figure 2.4. Showing the Options dialog (may need update)

The **Options** dialog is where all the global user preference parameters are defined, including the SCRIPT *Editor* preferences for syntax coloring, the default typeface of each report section, as well as, default data formats and other options. This dialog has the **Format**, **Data** and **Other** tabs. The **Format** tab contains the following controls:

Category list-box, which contains the following module category items:

- SQL Editor,
- SQL Result,
- Query Builder Editor,
- Query Builder Result,
- Data Shape Builder Editor,
- Data Shape Builder Result,
- SCRIPT Editor,
- Page Header,
- Page Footer,
- Report Header,
- Report Detail,

Report Footer,

Reset All button, which is used to reset all the options in this tab to the factory setting,
Choose Font button, which is used to set the typeface of the selected category via the **Font** dialog,
Font display field, which shows the selected font,
Size display field, which shows the selected typeface size,
Colors list-box, which is used to define the colors of syntax primitives, which are listed next:

Text,
Text Selection,
Number,
Tag Delimiter,
Comment,
Element,
String,
Attribute Name,
Constants,
Predefined Variable,
Keyword Tag,
Keyword Declaration,
Keyword Statement,
Control Type,
Data Type,

Foreground popup list-box, which is used to change the foreground color of elements defined by the selection in the **Colors** list-box,
Background popup list-box, which is used to change the background color of elements defined by the selection in the **Colors** list-box,
Sample display field, containing sample of text showing the effects of the selected options.

In particular, to change the syntax coloring configuration for the SCRIPT *Editor*, the user must select the SCRIPT Editor in the **Category** list-box, and then set the colors of the respective keywords or elements of the SCRIPT language. To change the typeface (or default typeface) of the selected category the user must click the **Font** button. This will call the **Font** dialog, in which the user may select the desired typeface, font style and font size.

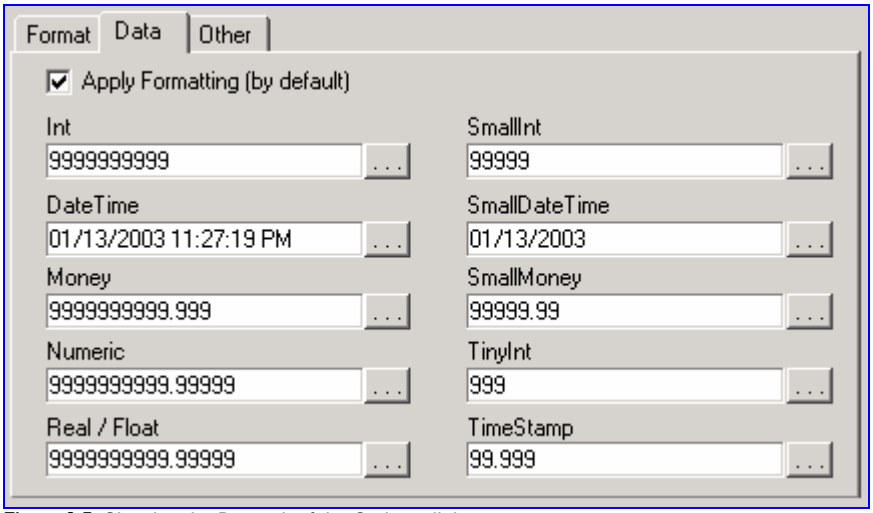


Figure 2.5. Showing the Data tab of the Options dialog

The **Data** tab of the **Options** dialog is where the default formatting for each data type is defined. This tab applies only to the following categories: Page Header, Page Footer, Report Header, Report Detail and Report Footer. You may disable the application of this automatic default formatting by making the **Apply Formatting (by default)** check-box unchecked. This action will disable all the controls in this tab, and will not use automatic formatting whenever data fields or expressions are inserted into the report.

If automatic formatting is enabled, the user may change the default format for any of the data types listed in the **Data** tab by clicking on the corresponding 3-dotted button. This will prompt the **Format/Conversion Wizard** dialog in which the user may set the desired format (see Figure 2.6 for more details). Depending on the selected

data type, the **Expression Type**, **Decimal Places** and **Format** control will be pre-selected. The user then may set the desired formatting by either selecting a standard format from the **Format** combo-box, or manually enter a valid format specification. We will describe the details of the `Format` function and the rules of format specification later on in this chapter. For now, all we need to use is the factory set default formatting, as shown in Figure 2.5 for each data type.

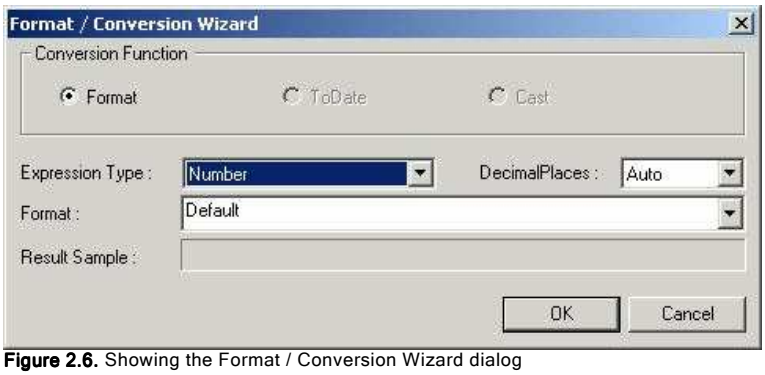


Figure 2.6. Showing the Format / Conversion Wizard dialog

The **Other** tab of the **Options** dialog contains some global options for the *Report Designer* application to control the behavior of the following features:

Enable Intellisense	check-box to enable/disable *IntelliSense* in the **Source** view and other related editors. By default this feature is enabled.
Make Source default view	check-box to set the **Source** view as the default view when the user initially opens a report script. By default the **Design** view is selected, but it is recommended to check this check-box for advanced reports.
Use ADO bound grid to display query results	check-box to use the ADO data-bound grid for the query result display. By default this check-box is checked.
Make query results directly editable	check-box to enable direct data editing from the query results. By default this check-box is unchecked.
Query command timeout	edit-box to adjust the default query command timeout in seconds. By default this value is 0, which means indefinite time or no timeout restrictions.

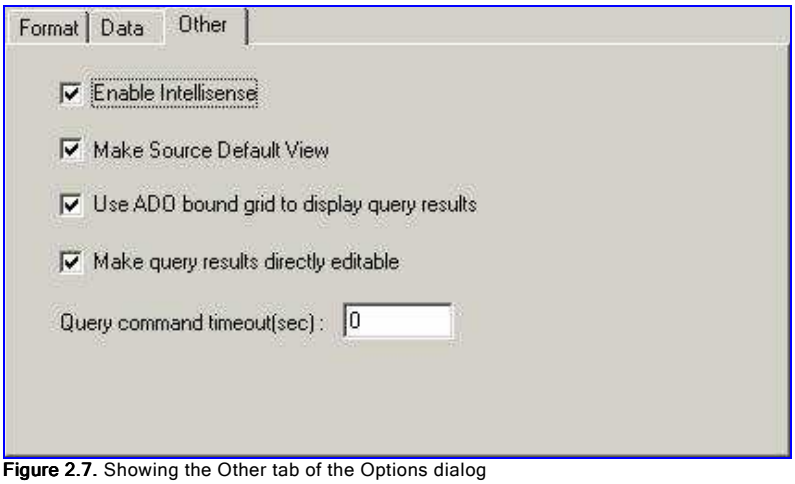


Figure 2.7. Showing the Other tab of the Options dialog

A Simple Example

In this section we will give an introductory tutorial on how to use the SCRIPT *Editor* to create and maintain a simple report. Along the way, we will outline details about various GUI windows and dialogs and their respective usage. In particular, we will describe details about the **Project Explorer** pane, the **Data View** pane, the **New Report** dialog, the **Report Settings** dialog, the **New Database Connection** dialog, the **Query Builder** window,

the **Insert Table** dialog, the **Data Fields** and **Expression Builder** windows.

We start by creating a new report project, which may be achieved by selecting the **New Project** menu item of the **File** menu, or alternatively clicking on the **New Project** toolbar button. This will display the **New Project** dialog, prompting the user for a project name. Enter a name for your first project in the **Project name** edit-box, for example type MyProj1. Observe that as you are typing this name, the **Location** edit-box will duplicate this name as a subdirectory to the current working directory, so that your report files will be located by default in the \MyProj1\REPORTS\ subdirectory, while the project file, MyProj1.RPJ, will be located by default in the \MyProj1\ subdirectory. After entering and submitting the **New Project** dialog, this new project is opened in the **Project Explorer** pane (see Figure 2.8).

Recall that the **Project Explorer** pane displays various elements that together form a collection of files used in the report module of your host application. In particular, it may include various SQL queries used in your reports, the corresponding database connections, the image and XML data file references, as well as, the SCRIPT report files themselves.

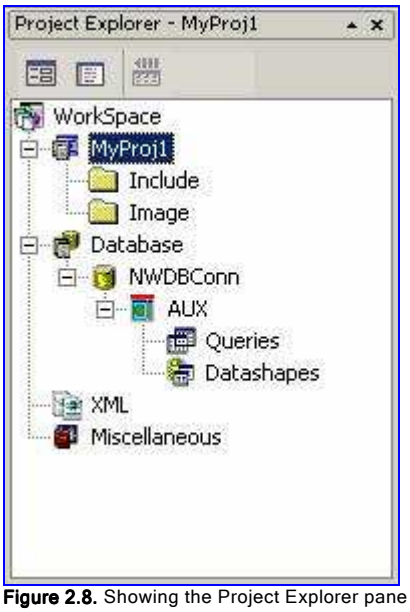


Figure 2.8. Showing the Project Explorer pane

A report project essentially consists of report and database files. The report files are essentially SCRIPT files with `.ELS` file extension. As we indicated that these files will be stored in the default \REPORTS subdirectory inside the project's root directory. Each database connection in the project will have a separate database file. These database files, which are files with `.AUX` file extension, will essentially contain all the queries and data shapes that are used in the project under that connection object.
In addition to these files, a report project may also contain SQL script files. These are essentially text files with the `.SQL` file extensions, presumably to store SQL scripts used in the project.
Observe that in the **Project Explorer** pane, the report files will be shown under the **MyProj1** root node, while the SQL and database files will be shown under the corresponding connection object in the **Database** root node.
The **Project Explorer** pane has a special toolbar, which consists of the **Design**, **Source** and **Parameter List** toolbar buttons. The **Design** and **Source** toolbar buttons will respectively open or display the selected report in **Design** or **Source** view, while the **Parameter List** buttons will slide on the **Parameter List** windows.

In the next few paragraphs we will use the popup menu of the **Project Explorer** pane to create a new report, as well as, new database connection and queries for the new report. We will then use the drag-drop capabilities of the pane to drop the defined queries into the report script.

To create a new report we click the right-mouse-button on the **MyProj1** node. This will open the **Project Explorer** pane's popup menu. From this menu select the **New** menu command, to call the **New Report** dialog, in which the user must enter the name of the new report and select the creation method.

Note that the **New Report** dialog has three radio-buttons for the different methods of report creation. These radio buttons along with other controls of the **New Report** dialog are listed next (see also Figure 2.9 for more details):

Report Name	edit-box, to enter the name of the report file,
Use Standard Template Wizard	radio-button, to select a template from the standard report template library, this button is selected by default,
Use Binary Report Template	radio-button, to select a binary report template *BRT*-file,
Custom	radio-button, to select an empty custom report template,
Cascade Style Sheet	frame, from which to select a *CSS* (i.e. *Cascade Style Sheet*).

To continue the creation of the new report, note that the **Use Standard Template Wizard** radio-button is selected by default. Leaving this option as is, enter a report name, say Report1, and then click the **OK** button to submit this dialog. The **Use Standard Template Wizard** radio-option selection will result in opening the **Select Standard Template** window, from which you must select an existing standard report template. On the right side of this

window, click on the **Custom** group arrow to open the folder. Depending on the content of your standard report template library, you must have a *BASIC01* template under this folder. Selecting this *BASIC01* template submit this window by clicking on the **OK** button.

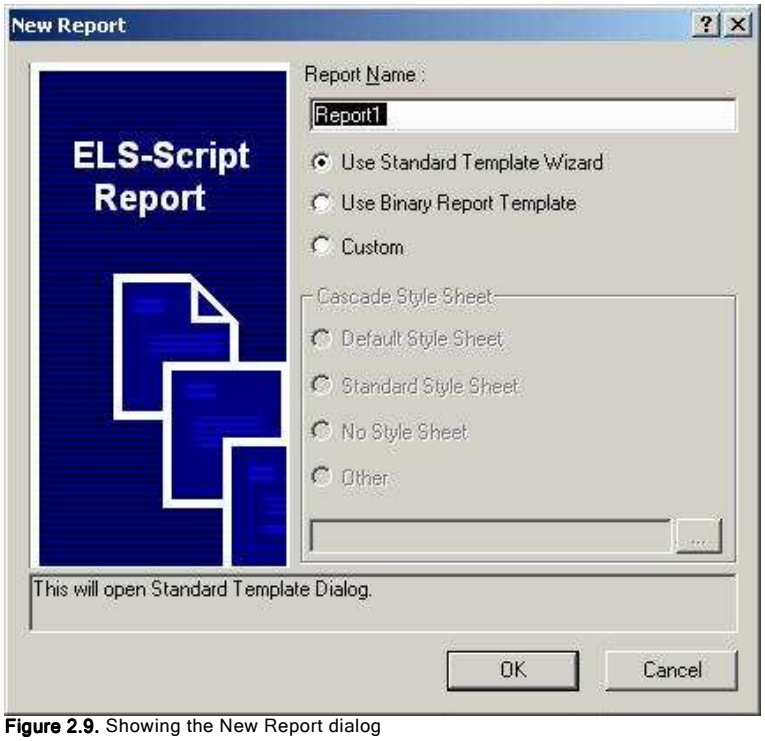


Figure 2.9. Showing the New Report dialog

The selection in the **Select Standard Template** window will call the *SRT Wizard* window with the *BASIC01* standard report template activated. In this standard report template wizard, note that you can modify the report title, the page orientation and size, the page margins, along with some basic elements. For now, we leave all the defaults as they are and click on the **OK** button of this window. This action will create the Report1.ELS report script and open it into the SCRIPT *Editor* window (i.e. **Source** view of the *Report Designer*, see Figure 2.10 for details).

Observe that the standard template that we selected already created the necessary report sections including the *ELS_RSETTINGS*, *ELS_QPARAMS*, *ELS_RHEADER*, *ELS_PHEADER*, *ELS_RDETAIL*, *ELS_PFOOTER* and *ELS_RFOOTER* sections. Observe also, that moat of these sections are imbedded inside the *HTML* `<BODY>` and `</BODY>` tags, with the `<HTML>` and *HTML* `<HEADER>` declarations at the beginning of the script document. So that anything outside pure SCRIPT element will be interpreted as HTML element.

In particular, note that the page header has the following script:

```
<ELS_PHEADER HEIGHT="30px" FONT-FAMILY="Arial" FONT-SIZE="8pt">
<TABLE style="FONT-SIZE: 8pt; WIDTH: 100%; HEIGHT: 20px">
<TBODY>
     <TR style="HEIGHT: 16px" vAlign=top>
           <TD style="FONT-WEIGHT: bold; WIDTH: 541px">
               <SPAN class=Field style="OVERFLOW: hidden; WIDTH: 408px; COLOR: gray;
                                        WHITE-SPACE: nowrap; HEIGHT: 14px">
               <FLD>"Report Title"</FLD></SPAN>
           </TD>
           <TD style="WIDTH: 173px; TEXT-ALIGN: right">
               <SPAN class=Field style="OVERFLOW: hidden; WIDTH: 141px; COLOR: gray;
                                        WHITE-SPACE: nowrap">
               <FLD>"P " + Format(PageNum(),"") + " / " + Format(PageCount(),"")</FLD></SPAN>
           </TD>
     </TR>
</TBODY>
</TABLE>
</ELS_PHEADER>
```

Particular attention must be given to the HEIGHT attribute used in the ELS_PHEADER tag. This restricts the height of the page header to the specified 30 pixels. Any object exceeding this specified height will be clipped by the excessive amount. Also, we should emphasize that attribute values of SCRIPT tags need to be specified between double quotes, and that unlike HTML interpreters, the SCRIPT compiler will not understand values specified without double quotes, and will generate compile-time errors (e.g. for example, error messages that - SCRIPT's prime directives have been violated with meaningless text, and that the user needs to negotiate on this matter before any report compilation attempt can be made).

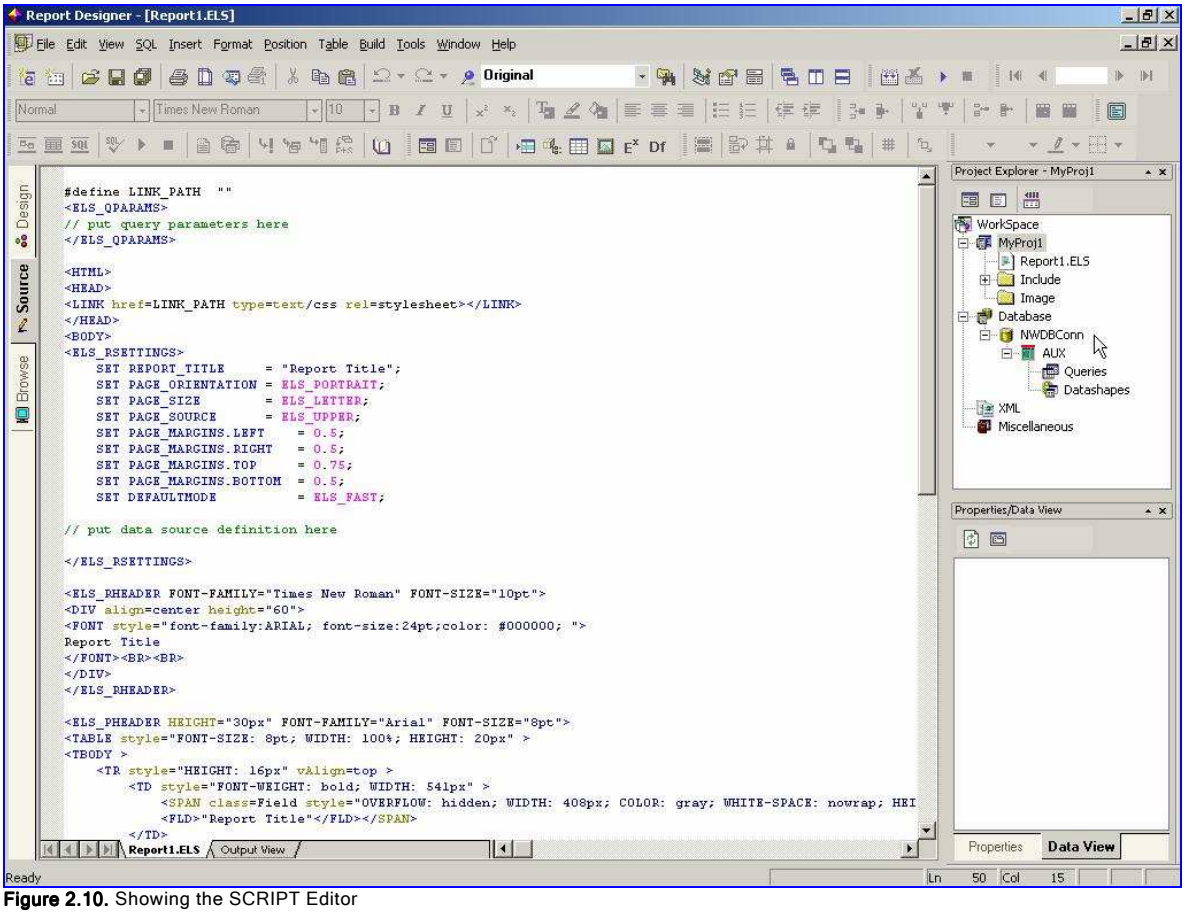


Figure 2.10. Showing the SCRIPT Editor

In this template, the page header essentially contains an *HTML Table*, which has a single row of two columns. The style of the <TABLE> tag has font size of 8pt, width of 100% of the BODY. The style of the first column has bold font, and the width of the column is 541px. The style of the second column inherits all attributes from the table style, and in addition, has right text alignment. The first column contains a *FLD*-element of a constant string for report title. The second column has a *FLD*-element containing the expression:

```
"P " + Format(PageNum(),"") + " / " + Format(PageCount(),"")
```

The PageNum() is the SCRIPT language function which returns the page number of the current page during the report generation. The PageCount() function on the other hand, returns the number of total pages after the whole report is generated. The Format function essentially converts the first argument's value into a string.

In a similar spirit, the page footer section of the report makes a use of the GetDate() function in a *FLD*-element, bounded by an *HTML* <SPAN> tags, as shown in the following code:

```
<ELS PFOOTER HEIGHT="30px" FONT-FAMILY="Arial" FONT-SIZE="8pt">
<TABLE style="FONT-SIZE: 8pt; WIDTH: 100%; HEIGHT: 20px">
<TBODY>
    <TR style="HEIGHT: 16px" vAlign="top">
       <TD style="FONT-WEIGHT: bold; WIDTH: 541px">
```

```
            <SPAN class=Field style="OVERFLOW: hidden; WIDTH: 408px; COLOR: gray;
                                     WHITE-SPACE: nowrap; HEIGHT: 14px">
            <FLD>"RUN DATE-TIME: " + Format(GetDate(),"mm/dd/yy") + "("
                                     + Format(GetDate(),"hh:nn:ss") + ")"</FLD>
            </SPAN>
        </TD>
        <TD style="WIDTH: 173px; TEXT-ALIGN: right">
            <SPAN class=Field style="OVERFLOW: hidden; WIDTH: 141px; COLOR: gray;
                                     WHITE-SPACE: nowrap">
            <FLD>" "</FLD>
            </SPAN>
        </TD>
    </TR>
</TBODY>
</TABLE>
</ELS_PFOOTER>
```

Again, note the `HEIGHT` attribute of the `<ELS_PFOOTER>` SCRIPT tag defining the height in pixels of the page footer section in the report output.

We next look at the *ELS_RSETTINGS* section of the report, which was filled in for us by the *BASIC01* standard report template:

```
<ELS_RSETTINGS>
    SET REPORT_TITLE      = "Report Title";
    SET PAGE_ORIENTATION  = ELS_PORTRAIT;
    SET PAGE_SIZE         = ELS_LETTER;
    SET PAGE_SOURCE       = ELS_UPPER;
    SET PAGE_MARGINS.LEFT        = 0.50;
    SET PAGE_MARGINS.RIGHT       = 0.50;
    SET PAGE_MARGINS.TOP         = 0.75;
    SET PAGE_MARGINS.BOTTOM      = 0.50;
    SET DEFAULTMODE              = ELS_FAST;
</ELS_RSETTINGS>
```

Recall that the *ELS_RSETTINGS* section allows only pure SCRIPT elements, and unlike the five report sections, it cannot contain segments that are mixed with HTML elements. The user may change or add report settings options via manual entry with the help of the **List Report Settings** menu item of the **Source** view's popup menu together with *IntelliSense*, or alternatively calling the **Report Settings** dialog. In this simple example, we will be using the **Report Settings** dialog to make some changes to the report title and page margins. To do so, we call this dialog via the **Report Settings** menu item under the **File** menu. The **Report Settings** dialog contains the following controls (see Figure 2.11 for further details):

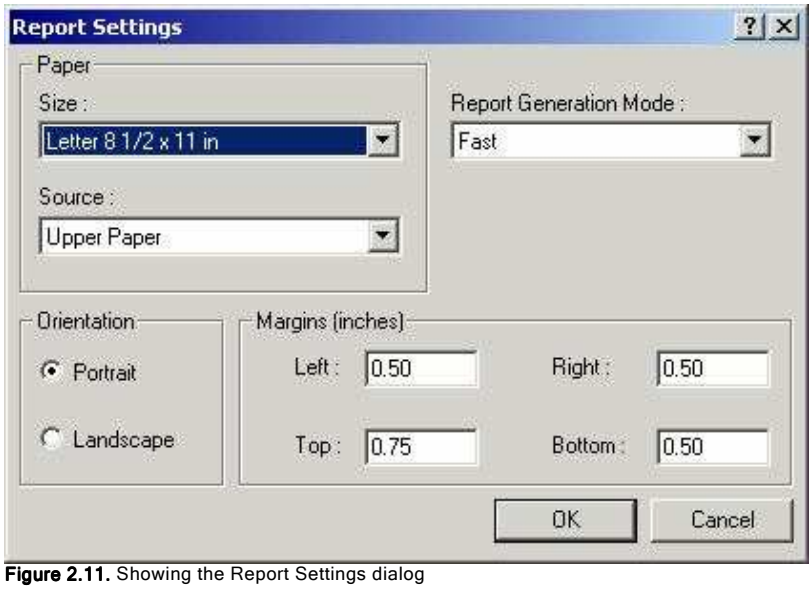


Figure 2.11. Showing the Report Settings dialog

Paper frame containing the following controls:
- **Size** combo-box, with a list containing all possible printer paper sizes,
- **Source** combo-box, with a list containing all possible printer paper sources,

Report Generation Mode combo-box, with a list containing the three generation modes of the SCRIPT engine: *Continuous*, *Fast* and *Style*,

Orientation frame containing the following controls:
- **Portrait** radio-button, to make report's page orientation portrait,
- **Landscape** radio-button, to make report's page orientation landscape,

Margins frame containing the following controls:
- **Left** edit-box, to set the left margin of the page in inches,
- **Right** edit-box, to set the right margin of the page in inches,
- **Top** edit-box, to set the top margin of the page in inches,
- **Bottom** edit-box, to set the bottom margin of the page in inches,

Where the default paper size is *Letter 8 ½ x 11 in*, default paper source is *Upper Paper*, default orientation is *Portrait*, and default report generation mode is *Fast*.

Observe in the **Margins** frame that the left, right and bottom margins are all set to 0.50 inches, and the top margin is set to 0.75 inches. The first change that we like to perform is to set the top margin to 1.00 inch. After performing this change and submitting the **Report Settings** dialog, the line in the *ELS_RSETTINGS* script representing the top margin will be changed to the following:

```
SET PAGE_MARGINS.TOP = 1.00;
```

The value of `REPORT_TITLE` variable will become the HTML title in the `<HEAD>` section of the report output, when considered as an HTML document. Since we will be using the *Orders* data from the *Northwind* database that comes with MS-SQL Server, we will modify this value to the text "`Orders Report`" by entering this value directly in the **Source** view. In fact, any of the values in the *ELS_RSETTINGS* section could have been modified by direct editing in the **Source** view.

We define next a default data access connection object for this report project. To define such a connection select the **New Database Connection** menu item from the **File** menu, or alternatively position the mouse pointer on the **Database** node of the **Project Explorer** pane and using the right mouse-button click to open the popup menu, and select **New** menu item from the popup menu. This will prompt the **New Database Connection** dialog for the connection object's name and data access type. Enter a desired name for the database connection object, say MyNWDB, and select the **OLE DB Dynamic Data Access** data access type, as shown in Figure 2.12:

Figure 2.12. Showing the New Database Connection dialog

After submitting this dialog, the **Data Link Properties** dialog appears with the **Provider** tab selected by default. In this tab, we should select the **Microsoft OLE DB Provider for SQL Server** (since we want to connect to the MS-SQL Server) and click the **Next** button, which will switch to the **Connection** tab. In this tab, select or enter your

MS-SQL Server name in *step 1*, enter your log on information in *step 2*, and select the *Northwind* database in *step 3*, then click the **OK** button. This will create a new connection node in the **Project Explorer** pane, under the **Database** parent node, with the name MyNWDB. Double click this MyNWDB connection object node to open the database into the **Data View** pane, as shown in Figure 2.13 below.

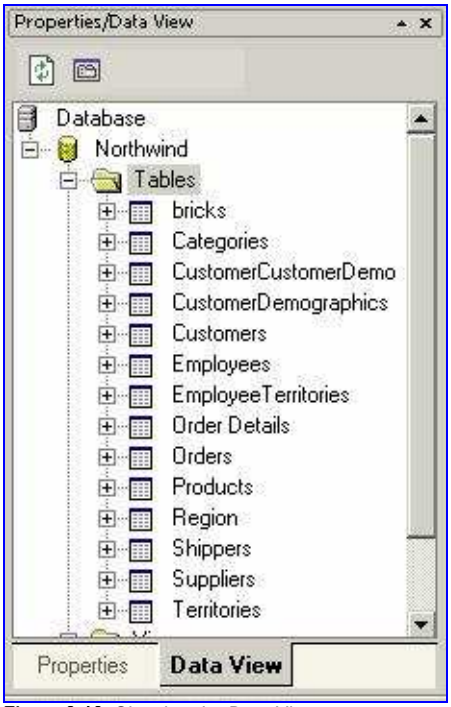


Figure 2.13. Showing the Data View pane

The **Data View** pane shows the structure of the tables, views stored-procedures and user-defined functions of the back-end database of the currently open connection object. In particular, you may use this pane to explore the fields of the tables, as well as, the parameter structure of the stored-procedures, that are shown in the pane. Moreover, you may use the **Get Details** menu item of the popup menu to get the actual script of the selected database object.

Data source object utilization methods in the *Report Designer* are as follows. We first create and perfect a data source in the *Query Builder* or the *Data Shape Builder* module, and when we are ready to use this data source, we need to save this object as a query or data shape node under the related database connection node. So that we can drag-drop this node into the report script. After insertion of such object, the inserted text can be freely edited, and will have no relation to the original saved query object in the **Project Explorer** pane.

We illustrate this method by creating a simple data source utilizing only the *Orders* table of the *Northwind* database. In the **Project Explorer** pane, expand the MyNWDB database connection node, and observe that it contains the **AUX** child node, which in turn contains two child nodes, namely, the **Queries** and **Datashapes** container nodes. Use the right-mouse-button click method over the **Queries** node to display the popup menu. From this popup menu select the **New** menu command. This will create a new instance of the query builder window opening it into the *Report Designer* application.

Recall from Chapter 1, that the query builder window consists of three panes, namely, the **Relations** pane, the **Columns** pane and the **SQL** pane, which work together to simplify the task of SQL command construction.

To continue the creation of the data source, simply drag-drop the *Orders* table from the **Data View** pane into the **Relations** pane (i.e. the topmost pane in the query builder window). This will make a diagram of the table inside the **Relations** pane, from which select the *OrderID*, *OrderDate*, *ShippedDate*, *ShipName* and *ShipAddress* fields.

At this point the **SQL** pane will contain the following text:

```
SELECT
        OrderID,
        OrderDate,
        ShippedDate,
        ShipName,
        ShipAddress
FROM
        Orders
```

Change the name of the NewQuery node just created in the **Project Explorer** to MyOrders, by clicking a second time on the node, which makes the node name editable (similar to *Windows Explorer*). Finally save the changes you have made to this query by selecting the **Save** menu item from the **File** menu (or equivalently by clicking the **Save** toolbar button). You may want to test this query by executing it via the **Execute Query** green arrow button in the **SQL** main toolbar, which will display all records prescribed by the SQL command defining the query. For more details about creating and maintaining queries consult the query builder online help via the **Query Builder Help** menu command of the **Help** menu, for now you may close the newly created query.

Now to put the data source into the Report1.ELS script, make some empty lines just before the `</ELS_RSETTINGS>` tag, and drag the MyOrders query node and drop it on this empty line. This action will result to the following insertion:

```
<ELS_RSETTINGS>
     ...
     SET DEFAULTMODE = ELS_FAST;
     // put data source definition here

     DECLARE @MyOrders DATASOURCE;
     SET @MyOrders = "SELECT " +
                             "OrderID, " +
                             "OrderDate, " +
                             "ShippedDate, " +
                             "ShipName, " +
                             "ShipAddress " +
                     "FROM " +
                             "Orders";
</ELS_RSETTINGS>
```

Note that the drag-drop action, essentially declared a variable of type `DATASOURCE`, using the name of the query node as the variable name. It also, translated the SQL command into a valid string concatenation and set the variable to this string expression. As you can see, this single drag-drop operation may save the user a lot of tedious coding work, especially when the query text is several lines long. To check the correctness or simply to view the existing data sources in the current report, you may use the **Connection / Datasource List** window via the corresponding menu item under the **Tools** menu.

We proceed next, to insert an *HTML Table* into the *ELS_RDETAIL* section of the report script. To do so, first position the cursor on the empty line preceding the `</ELS_RDETAIL>` tag, and then call the **Insert Table** dialog via the **HTML Table** menu item of the **Insert** menu (or equivalently via the **Insert HTML Table** toolbar button). In this **Insert Table** dialog, set the **Rows** to 1, **Columns** to 5, in the **Table attributes** set **Width** to 100 percent, and finally set the **Border size** to 1 pixels (see Figure 2.14 for more details).

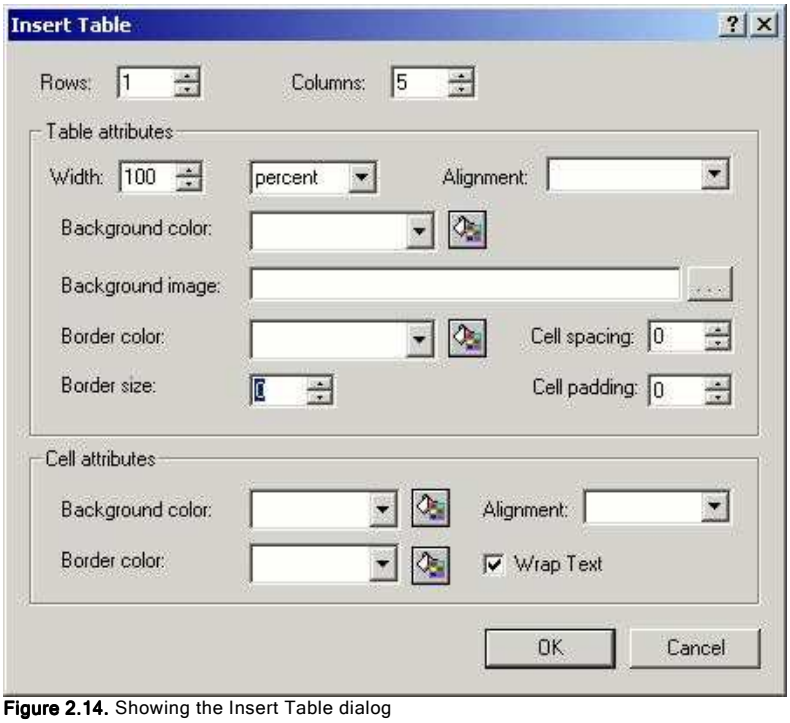


Figure 2.14. Showing the Insert Table dialog

Submitting this dialog will insert *HTML Table* script into the report script at the cursor location, so that the *ELS_RDETAIL* section becomes as follows:

```
<ELS_RDETAIL>
// put report detail information here
<TABLE border=1 CellSpacing=0 CellPadding=0 WIDTH="100%">
```

```
    <TR>
            <TD></TD>
            <TD></TD>
            <TD></TD>
            <TD></TD>
            <TD></TD>
    </TR>
</TABLE>
</ELS_RDETAIL>
```

We are now ready to insert data fields into the columns of this table, therefore we insert an empty line between each `<TD>` and `</TD>` tags, and then call the **Data Fields** window via the **Data Fields** menu item of the **Insert** menu. The **Data Fields** window will display all the existing data sources used in the report under the **Fields** tree-view. In particular, you will see the `@MyOrders` data source variable. Click on the plus icon to expand this node into the field nodes. Then drag fields one by one and drop them in between each `<TD>` and `</TD>` tag pairs, as shown in Figure 2.15 below:

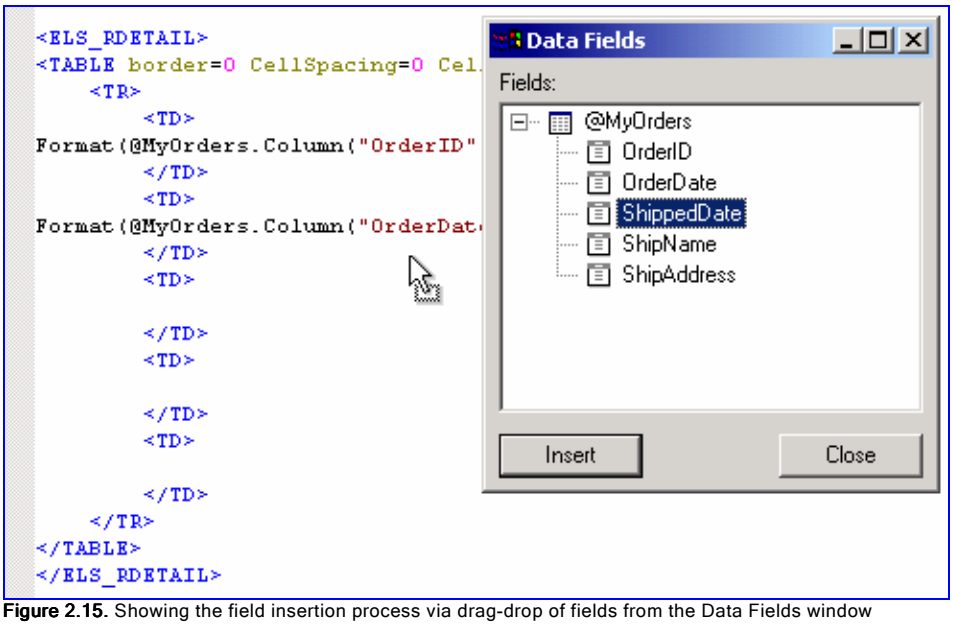


Figure 2.15. Showing the field insertion process via drag-drop of fields from the Data Fields window

So that the five fields *OrderID*, *OrderDate*, *ShippedDate*, *ShipName* and *ShipAddress*, are respectively inserted into the five `<TD>` and `</TD>` tag pairs.
After insertion, observe that the SCRIPT's `Format` function is applied over the first three fields. In particular, the date type fields *OrderDate* and *ShippedDate* have default format specification "`mm/dd/yyyy hh:nn:ss AM`". Modify this specification for both fields to include only the month-day-year information, so that the *ELS_RDETAIL* section becomes as follows:

```
<ELS_RDETAIL>
<TABLE border=1 CellSpacing=0 CellPadding=0 WIDTH = "100%">
    <TR>
            <TD>
            <FLD>Format(@MyOrders.Column("OrderID"), "")</FLD>
            </TD>
            <TD>
            <FLD>Format(@MyOrders.Column("OrderDate"), "mm/dd/yyyy")</FLD>
            </TD>
            <TD>
            <FLD>Format(@MyOrders.Column("ShippedDate"), "mm/dd/yyyy")</FLD>
            </TD>
            <TD>
            <FLD>@MyOrders.Column("ShipName")</FLD>
            </TD>
            <TD>
            <FLD>@MyOrders.Column("ShipAddress")</FLD>
            </TD>
```

```
    </TR>
</TABLE>
</ELS_RDETAIL>
```

So far, what we have constructed is only the record structure of a tabular presentation of the MyOrders data. We still need iteration control to present all the records of this data source. This necessitates the use of pure SCRIPT syntax sections utilizing the <ELS> and </ELS> tags. In particular, we need to add the following code just after the <ELS_RDETAIL> tag:

```
    <ELS>
    WHILE NOT @MyOrders.Eof()
    </ELS>
```

and the following code just after the </TABLE> tag, but before the </ELS_RDETAIL> tag:

```
    <ELS>
            @MyOrders.Next();
    END LOOP
    </ELS>
```

So that the entire *ELS_RDETAIL* section now becomes as follows:

```
<ELS_RDETAIL>
<ELS>
WHILE NOT @MyOrders.Eof()
</ELS>
<TABLE border=1 CellSpacing=0 CellPadding=0 WIDTH = "100%">
    <TR>
            <TD>
            <FLD>Format(@MyOrders.Column("OrderID"), "")</FLD>
            </TD>
            <TD>
            <FLD>Format(@MyOrders.Column("OrderDate"), "mm/dd/yyyy")</FLD>
            </TD>
            <TD>
            <FLD>Format(@MyOrders.Column("ShippedDate"), "mm/dd/yyyy")</FLD>
            </TD>
            <TD>
            <FLD>@MyOrders.Column("ShipName")</FLD>
            </TD>
            <TD>
            <FLD>@MyOrders.Column("ShipAddress")</FLD>
            </TD>
    </TR>
</TABLE>
<ELS>
    @MyOrders.Next();
END LOOP
</ELS>
</ELS_RDETAIL>
```

There are a few more steps to make this report more acceptable from the viewpoint of good presentation, but we think it is time to try compiling and running this report in the *Report Designer*. To compile the report click the **Compile ELS File** toolbar button (or equivalently select the **Compile** menu command from the **Build** menu). A result pane will appear at the bottom of the **Source** view window showing the compilation process, and if there are no errors, the message will be as follows (displayed in the **Build** tab of the **Results** pane):

```
Compiling...
Report1.ELS
Linking...

0 error(s), 0 warning(s)
```

If there were errors, the SCRIPT compiler would have indicated the nature and location of the error in the script. Moreover, a double-click on the error message line in the **Results** pane, will make the **Source** view jump to that

error location in the script. After a successful compilation, you are ready to run the report by clicking the **Run** toolbar button. This action will generate the report output listing the pages in the **Results** pane and displaying the first page in the **Output View** tab of the SCRIPT *Editor*. After the entire report generation, you may navigate through the pages of the report output using the **First**, **Previous**, **Next** and **Last** toolbar buttons.

Looking at the first page of the output, the first noticeable problem is the abnormal alignment of the field columns (see Figure 2.16 for more details).

Figure 2.16. Showing the abnormal alignment problem in the report output result

This problem steams from the HTML table render mechanisms, which tries to make the best suitable calculations for the width of the columns in the HTML table depending on the size of the content of the column. And since each line of the output is itself an HTML table, this alignment variation will occur from one line to the other. But nevertheless, the HTML standards are so powerful that this problem becomes mere feature and not at all a problem. As we will discover that adding some width attributes to the `<TD>` columns of the HTML table will resolve this alignment problem. More precisely, we must set some percentage widths to the `<TD>` tags, for example, 10%, 20%, 40%, etc. So that the code becomes as follows:

```
<ELS_RDETAIL>
<ELS>
WHILE NOT @MyOrders.Eof()
</ELS>
<TABLE border=1 CellSpacing=0 CellPadding=0 WIDTH = "100%">
    <TR>
            <TD WIDTH = "10%">
            <FLD>Format(@MyOrders.Column("OrderID"), "")</FLD>
            </TD>
            <TD WIDTH = "10%">
            <FLD>Format(@MyOrders.Column("OrderDate"), "mm/dd/yyyy")</FLD>
            </TD>
            <TD WIDTH = "10%">
            <FLD>Format(@MyOrders.Column("ShippedDate"), "mm/dd/yyyy")</FLD>
            </TD>
            <TD WIDTH = "30%">
            <FLD>@MyOrders.Column("ShipName")</FLD>
            </TD>
            <TD WIDTH = "40%">
            <FLD>@MyOrders.Column("ShipAddress")</FLD>
            </TD>
    </TR>
</TABLE>
<ELS>
```

```
    @MyOrders.Next();
END LOOP
</ELS>
</ELS RDETAIL>
```

Alternatively, a more indepth study of the HTML *SPAN*-tag element reveals some very appropriate style properties, namely the `OVERFLOW` and `WHITE-SPACE` properties. These properties along with the `WIDTH` and `HEIGHT` style properties will control both the horizontal and vertical alignments in a much nicer way. Therefore, we need to wrap the *FLD*-elements in the following manner:

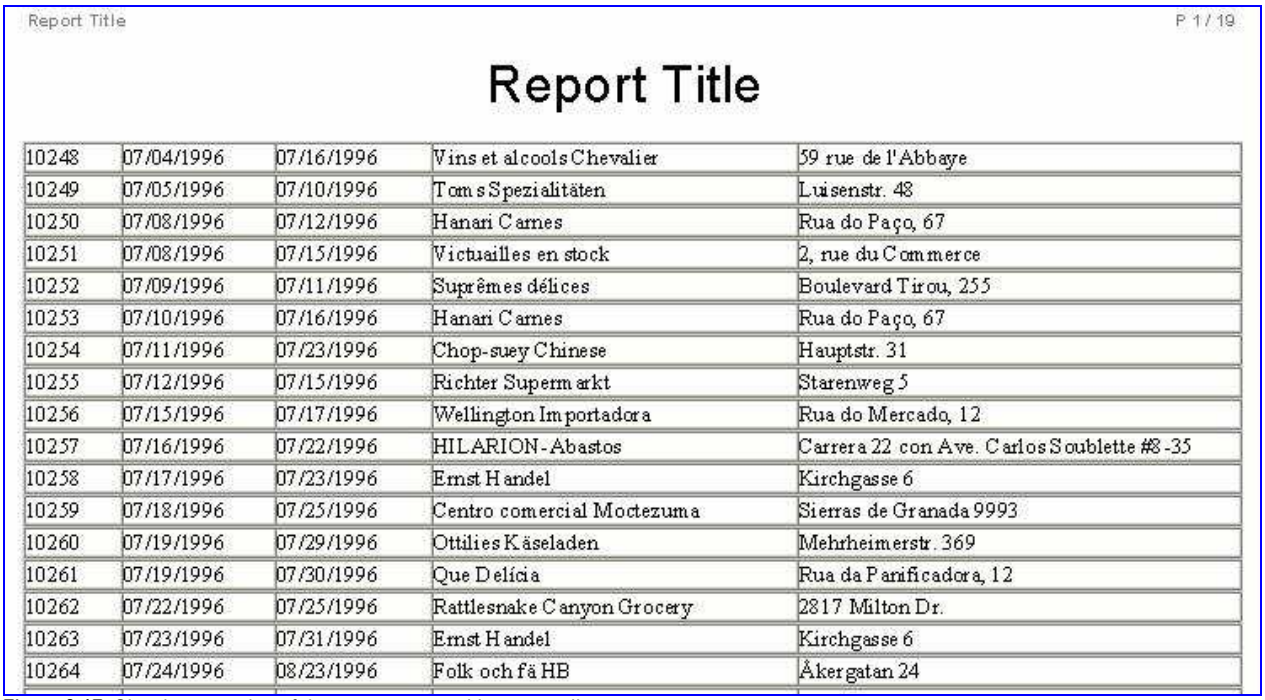
```
    <SPAN style="OVERFLOW:hidden;WHITE-SPACE:nowrap;WIDTH:width_value">
    <FLD>...</FLD>
    </SPAN>
```

where *width_value* is an explicit width size of the column in specific units (e.g. `0.5in`, `20px` or `10pt`). For example, after adding these tags with appropriate width values, we may get the following code for the *ELS_RDETAIL* section:

```
<ELS_RDETAIL>
<ELS>
WHILE NOT @MyOrders.Eof()
</ELS>
<TABLE border=1 CellSpacing=0 CellPadding=0 WIDTH = "100%">
    <TR>
            <TD>
            <SPAN style="OVERFLOW:hidden;WHITE-SPACE:nowrap;WIDTH: 52px">
            <FLD>Format(@MyOrders.Column("OrderID"), "")</FLD>
            </SPAN>
            </TD>
            <TD>
            <SPAN style="OVERFLOW:hidden;WHITE-SPACE:nowrap;WIDTH: 84px">
            <FLD>Format(@MyOrders.Column("OrderDate"), "mm/dd/yyyy")</FLD>
            </SPAN>
            </TD>
            <TD>
            <SPAN style="OVERFLOW:hidden;WHITE-SPACE:nowrap;WIDTH: 84px">
            <FLD>Format(@MyOrders.Column("ShippedDate"), "mm/dd/yyyy")</FLD>
            </SPAN>
            </TD>
            <TD>
            <SPAN style="OVERFLOW:hidden;WHITE-SPACE:nowrap;WIDTH: 204px">
            <FLD>@MyOrders.Column("ShipName")</FLD>
            </SPAN>
            </TD>
            <TD>
            <SPAN style="OVERFLOW:hidden;WHITE-SPACE:nowrap;WIDTH: 246px">
            <FLD>@MyOrders.Column("ShipAddress")</FLD>
            </SPAN>
            </TD>
    </TR>
</TABLE>
<ELS>
    @MyOrders.Next();
END LOOP
</ELS>
</ELS RDETAIL>
```

Recompiling the report after these changes and running it will result into correct alignment of the field columns, as is demonstrated in Figure 2.17.

Incidentally, it is a good place to point out that HTML table and SCRIPT *FLD*-element resize operations in the **Design** view of the *Report Designer* will essentially amount to such wrapping of *SPAN*-tags around each resized *FLD*-element. Therefore, there is no need to manually perform this *SPAN-wrap* operation in the **Source** view, especially when appropriate column widths in pixels or points may become time consuming calculation. In such a situation, the user may simply switch from the **Source** to **Design** view and visually resize each *FLD*-element along with the container cells. Recall also, that for such resize operations, the **Precision Resizer** is an impressive tool.



10248	07/04/1996	07/16/1996	Vins et alcools Chevalier	59 rue de l'Abbaye
10249	07/05/1996	07/10/1996	Toms Spezialitäten	Luisenstr. 48
10250	07/08/1996	07/12/1996	Hanari Carnes	Rua do Paço, 67
10251	07/08/1996	07/15/1996	Victuailles en stock	2, rue du Commerce
10252	07/09/1996	07/11/1996	Suprêmes délices	Boulevard Tirou, 255
10253	07/10/1996	07/16/1996	Hanari Carnes	Rua do Paço, 67
10254	07/11/1996	07/23/1996	Chop-suey Chinese	Hauptstr. 31
10255	07/12/1996	07/15/1996	Richter Supermarkt	Starenweg 5
10256	07/15/1996	07/17/1996	Wellington Importadora	Rua do Mercado, 12
10257	07/16/1996	07/22/1996	HILARION-Abastos	Carrera 22 con Ave. Carlos Soublette #8-35
10258	07/17/1996	07/23/1996	Ernst Handel	Kirchgasse 6
10259	07/18/1996	07/25/1996	Centro comercial Moctezuma	Sierras de Granada 9993
10260	07/19/1996	07/29/1996	Ottilies Käseladen	Mehrheimerstr. 369
10261	07/19/1996	07/30/1996	Que Delícia	Rua da Panificadora, 12
10262	07/22/1996	07/25/1996	Rattlesnake Canyon Grocery	2817 Milton Dr.
10263	07/23/1996	07/31/1996	Ernst Handel	Kirchgasse 6
10264	07/24/1996	08/23/1996	Folk och fä HB	Åkergatan 24

Figure 2.17. Showing a section of the report output with correct alignment

For this simple report there are two more additional steps that we can take to make it look like a real report. First, we can add a cover page with title and image, and secondly, it would be nice if we can add column headers.

Given that the report header section is a suitable place where one can put a cover page, we add a title and an image in the *ELS_RHEADER* section, so that we will have the following code:

```
<ELS_RHEADER>
<BR><BR>
<DIV style="FONT-FAMILY:arial; FONT-WEIGHT: bold; FONT-SIZE: 26pt;
                          COLOR:gray; TEXT-ALIGN: center">
// put an HTML horizontal line
<HR>
My First SCRIPT Report <BR>
Northwind Orders
<HR>
<BR><BR>
// put an image of BMP, JPG, GIF or any other format
<IMG height=364 width=512 hspace=0 align=baseline border=0
           src="THIS_FILE\Image\Stonehenge.jpg">
</DIV>
// this is the SCRIPT's way of forcing a page break
<ELS_PB>
</ELS_RHEADER>
```

In this code, we have used an HTML *DIV*-element, with a style specification of bold font of font-size 26 points, and a centered text alignment for the two lined title "`My First SCRIPT Report`" and "`Northwind Orders`". This text starts and ends with horizontal lines (check for the `<HR>` element). Following these, we have inserted an image from the \Image subdirectory of this project. Note that `THIS_FILE` is an SCRIPT keyword and refers to the current working directory path. We emphasize also, that height and width values must be specified for images either as attribute or style, otherwise the picture may get clipped. Finally, this *DIV*-element is followed with the page break keyword `<ELS_PB>` of the SCRIPT language.

We should point out that the image reference in this SCRIPT code segment assumes that an image file with the name Stonehenge.JPG must exist in the \REPORTS\Image subfolder of the project. This means that the user must copy all image files that are used in a report into this \REPORTS\Image subfolder. As a matter of fact, this is necessary only when the user enters such *IMG*-element via direct editing in the **Source** view. To automatically

copy images into the \REPORTS\Image subfolder, use the **Insert Image** dialog via the **Picture** menu command of the **Insert** menu.

Although the SCRIPT language has special functions to manipulate nested column headers, nevertheless in this section we will use the page-begin event to construct the column header from scratch. To proceed in this direction, use the right mouse-button click to open the **Source** view's popup menu. From the **Add/Move To Section** menu select the **OnBeginPage** submenu item, this will create the page-begin event handler at the end of the script, so that one can define all actions that must be perform whenever during the report generation a new page starts. In our case, we just want to put a column header row at the beginning of every new page. Moreover, this row must have the same width properties as any record row. Therefore, using the row structure of the *ELS_RDETAIL* we copy and insert it in the event handler, then modify the code to get the following:

```
Event OnBeginPage
<ELS>
    IF PageNum() > 1 THEN
</ELS>
<TABLE cellSpacing=0 cellPadding=0 width="100%" border=1>
    <TR>
            <TD bgcolor="lightgrey">
            <SPAN style="OVERFLOW: hidden; WIDTH: 53px; WHITE-SPACE: nowrap">
            OrderID
            </SPAN>
            </TD>
            <TD bgcolor="lightgrey">
            <SPAN style="OVERFLOW: hidden; WIDTH: 84px; WHITE-SPACE: nowrap">
            Order Date
            </SPAN>
            </TD>
            <TD bgcolor="lightgrey">
            <SPAN style="OVERFLOW: hidden; WIDTH: 85px; WHITE-SPACE: nowrap">
            Shipped Date
            </SPAN>
            </TD>
            <TD bgcolor="lightgrey">
            <SPAN style="OVERFLOW: hidden; WIDTH: 203px; WHITE-SPACE: nowrap">
            Ship Name
            </SPAN>
            </TD>
            <TD bgcolor="lightgrey">
            <SPAN style="OVERFLOW: hidden; WIDTH: 246px; WHITE-SPACE: nowrap">
            Ship Address
            </SPAN>
            </TD>
    </TR>
</TABLE>
<ELS>
    END IF
</ELS>
End Event
```

Examining this code segment, observe that the inside *ELS*-tags a conditional check is being made to see if the page number is not the first page (i.e. the cover page), if not then a column header row is put at the beginning of the page. Note also, that event sections are mixed language sections, so you can include SCRIPT with HTML or other scripting languages.

Recompiling and running the report after these modifications, we will see that a bold column header row is added at the beginning of each page of the report output, and that the cover page will be excluded from this column header addition. Looking at the cover page though, we see that the report title and page information appear on the top of the page. It would be nice if we can exclude this from the cover page as well. To do this, add the following lines to the *ELS_RSETTINGS* report section:

```
    SET SUPPRESS_PHEADER.FIRSTPAGE = TRUE;
    SET SUPPRESS_PFOOTER.FIRSTPAGE = TRUE;
```

This report settings options will simply suppress the page header and page footer in the first page. The first two pages of the report output are shown in Figure 2.18 below:

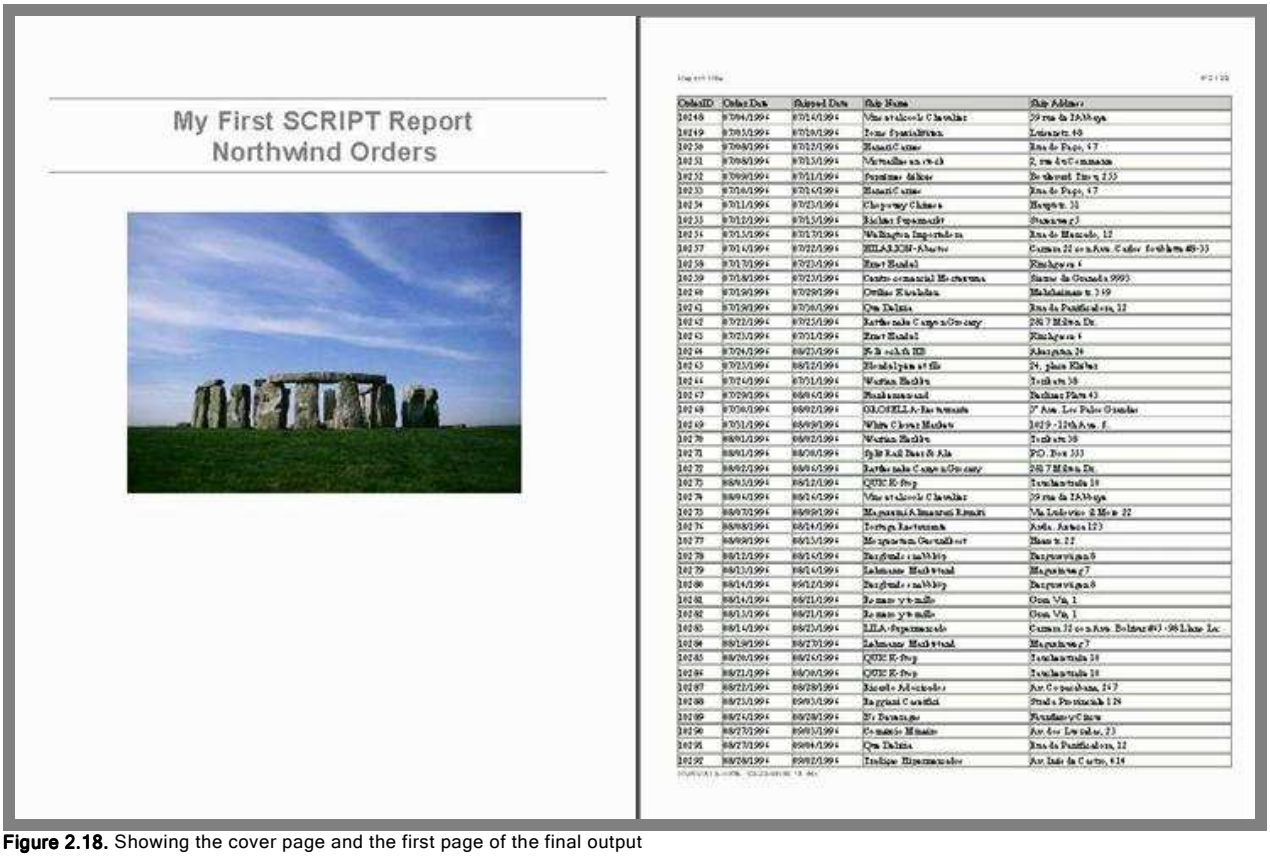


Figure 2.18. Showing the cover page and the first page of the final output

Advanced Syntax Elements

We hope that you have enjoyed the basics of the SCRIPT language, and that now you are ready to learn some advanced features of ELS-Script.

In this section, we will describe the rest of the syntax elements of the SCRIPT language, including the following features:

- More details about the *Connection* and *Datasource* objects
- Tabulation via the *ELS-Row* tags and *ResultRow* function
- The *Format* function and rules of format specification
- Conversion functions, including *ToDate* and *CAST*
- List of all functions and defining macros
- Specifying and using *Parameter Options*
- Specifying and using *Query Options*
- Pending variables and events
- More details about report settings

Connection and Datasource

In some cases it may be necessary to define data access within the report script itself rather than be passed from the host application. For such a situation, SCRIPT language provides the `CONNECTION` object, which essentially comes with all the capabilities that a data access may need, and can be imbedded directly inside the report script.

Furthermore, such an object may give the report developer the flexibilities to define multiple data access connections in a single report script. So that, for example, one may create a report, which collects information from various databases on various servers and presents a summary report.

Just like any other variable, a CONNECTION object variable must be explicitly declared before definition and usage. For example, the following code snippet declares and sets the connection string to a CONNECTION object variable:

```
DECLARE @Conn1 CONNECTION;

SET @Conn1 = "Provider=SQLOLEDB.1;Persist Security Info=False;User ID=sa;Data Source=HRSvr01";
```

Note that similar to the DATASOURCE object, a CONNECTION object variable establishes connection whenever the variable is set. Therefore, in the above code snippet the connection defined by @Conn1 becomes open and a connection is established with the HRSvr01 server.

We should emphasize that there is no need to worry about how to construct the connection string when setting a CONNECTION object variable. In fact, a drag-drop operation of any existing connection object node from the **Project Explorer** pane into the SCRIPT *Editor* will automatically insert the declaration and the definition of a connection object with full connection string already defined for you.

Connection objects have the following properties or methods:

TIMEOUT	this is the property to indicate, in seconds, how long to wait while executing a command before terminating the attempt and generating an error. Default value is 30 seconds, and it must be assigned to integer values only.
EXECUTE(sSQL)	this is the function to execute the SQL command specified by the sSQL argument. In particular, the command may be a valid SQL statement or a stored-procedure call. It returns a variant data type depending on the argument, if in particular, the argument is a row-returning query then the return will be a DATASOURCE object.

We next illustrate the usage of these syntax elements in the following code sample:

```
// declare connection variables and other necessary variables
DECLARE @connAccess, @connSQL CONNECTION;
DECLARE @ds DATASOURCE;
DECLARE @sAccessDS VARCHAR(300);
DECLARE @sUserID, @sDB, @sServer, @sPassword VARCHAR(50);
DECLARE @sSQL VARCHAR(1000);

// define the variable values, in a real-life situation these values may be
// passed from the host application via parameter or query options
SET @sAccessDS = "C:\Program Files\Microsoft Office\Office\Samples\Northwind.mdb";
SET @sUserID = "sa";
SET @sDB = "Northwind";
SET @sServer = "MyServer";
SET @sPassword = "my8739";

// define the connection strings
SET @connAccess = "Provider=Microsoft.Jet.OLEDB.3.15;Persist Security Info=False;" +
                  "Data Source=" + @sAccessDS;
SET @connSQL = "Provider=SQLOLEDB.1;Persist Security Info=False;" +
                  "User ID=" + @sUserID + ";" +
                  "Initial Catalog=" + @sDB + ";" +
                  "Data Source=" + @sServer + ";" +
                  "Password=" + @sPassword;

// change the default timeouts to new values,
// observe the keyword SET was not used since these are property assignments
@connAccess.TIMEOUT = 50;
@connSQL.TIMEOUT = @connAccess.TIMEOUT + 30;

// define the SQL command
SET @sSQL = "SELECT LastName, FirstName, Title, HireDate " +
```

```
            "FROM Employees " +
            "WHERE DATEPART(yy,HireDate)>=" + Format(Year(GetDate(), "") +
            " AND DATEPART(mm,HireDate)>=" + Format(Month(GetDate(), "");

// apply the EXECUTE method and since the defined command returns records
// we may set a data source variable to this returned object
SET @ds = @connAccess.EXECUTE(@sSQL);
```

So far we have illustrated the use of `TIMEOUT` property, as well as, the `EXECUTE` method for the special case when the SQL command is a *SELECT* statement. We will now continue this code sample, applying the `EXECUTE` method over a stored-procedure (for example the `[Employee Sales By Country]` stored-procedure of the MS-SQL Server's *Northwind* database), and then we will consider other SQL commands such as *INSERT* statements.

```
DECLARE @dtBeginDate, @dtEndDate DATETIME;
DECLARE @sSP VARCHAR(100);
DECLARE @dsX DATASOURCE;

SET @dtBeginDate = DateAdd(month, -3, GetDate());
SET @dtEndDate = GetDate();
// define the stored-procedure call command
SET @sSP = "exec [Employee Sales By Country] '" + Format(@dtBeginDate, "mm/dd/yyyy") +
           "','" + Format(@dtEndDate, "mm/dd/yyyy") + "'";
// we apply the EXECUTE method, note that this command also returns records
SET @dsX = @connSQL.EXECUTE(@sSP);
```

Here is an example of *INSERT* statement:

```
DECLARE @sLastName, @sFirstName, @sTitle VARCHAR(30);
DECLARE @dtHireDate DATETIME;

SET @sLastName = "Johns";
SET @sFirtName = "Barnaby";
SET @sTitle = "Manager, Sales";
SET @dtHireDate = ToDate("March 20, 2001", "mmmm d, yyyy");

SET @sSQL = "INSERT INTO employees (lastname, firstname, title, hiredate) " +
            "VALUES ('" + @sLastName + "','" + @sFirstName + "','" +
            @sTitle + "','" + Format(@dtHireDate,"mm/dd/yyyy") + "')";
// now apply the EXECUTE method, which will insert
// a record into the Employees table
@connSQL.EXECUTE(@sSQL);
```

Observe that in this code sample, we have made use of several SCRIPT functions, such as: `Format`, `GetDate`, `DateAdd` and `ToDate` functions. As we have indicated earlier that we will describe these functions, as well as, all other SCRIPT functions, in the later sections of this chapter. Another observation can be made concerning the applicability of the multiple connection feature of the SCRIPT language inside a single script. In fact, it is not very difficult to see that one may use this feature to create a data transformation from one database to another. For example, the following additional code will give us such a transformation from MS-Access *Northwind* database into MS-SQL Server *Northwind* database:

```
DECLARE @nCount INT;

// recall that @ds was assigned to the MS-Access Northwind
// recordset returned from Employees table
WHILE NOT @ds.EOF()
    SET @dtHireDate = @ds.Column("HireDate");
    SET @sFirstName = @ds.Column("FirstName");
    SET @sLastName = @ds.Column("LastName");
    SET @sTitle = @ds.Column("Title");
    SET @sSQL = "INSERT INTO employees (lastname, firstname, title, hiredate) " +
           "VALUES ('" + @sLastName + "','" + @sFirstName + "','" + @sTitle +
           "','" + Format(@dtHireDate, "mm/dd/yyyy") + "')";

    @connSQL.EXECUTE(@sSQL);
```

```
    SET @nCount = @nCount + 1;
    @ds.Next();
END LOOP
```

We consider next some more details concerning the DATASOURCE objects. Recall that DATASOURCE objects were described in "**Datasource Object and FLD-tag**" section of this chapter. The only difference is that here we illustrate the use of the Connect method, as well as, an alternative way of utilizing the Column method. The syntax of these DATASOURCE object's methods are as follows:

`Connect(CONNECTION objConn)`	function to define the data connection for the data source, if this function is never called for a particular DATASOURCE object, then the default connection of the report engine will be assumed. The argument must be an object of CONNECTION type (which will be described later in this chapter).
`Column(VARCHAR sFieldName)`	function to return the value of a particular field of the data source, returns a data type depending on the data type of the field in the backend database server. The argument must be a VARCHAR string representing a valid name of the field or alias of the field.
`Column(INT nFieldIndex)`	function to return the value of a particular field of the data source, returns a data type depending on the data type of the field in the backend database server that correspond to the specified field index. The argument must be an integer index corresponding to the 0-based index of the fields of the data source.

Therefore now the user may retrieve field values via the field/alias name, or alternatively via the 0-based index of the field in the definition of the data source. So that for example the four lines:

```
    SET @dtHireDate = @ds.Column("HireDate");
    SET @sFirstName = @ds.Column("FirstName");
    SET @sLastName = @ds.Column("LastName");
    SET @sTitle = @ds.Column("Title");
```

of the last code snippet can be alternatively performed via the following four lines:

```
    SET @dtHireDate = @ds.Column(3);     // since HireDate has index 3
    SET @sFirstName = @ds.Column(1);     // since FirstName has index 1
    SET @sLastName = @ds.Column(0);      // since LastName has index 0
    SET @sTitle = @ds.Column(2);         // since Title has index 2
```

Finally, the following code snippet illustrates the usage of the Connect method:

```
// assuming the connection object of earlier sample code
DECLARE @dsNew DATASOURCE;

@dsNew.Connect(@connSQL);
SET @dsNew = "SELECT * FROM employees";
```

We should emphasize that if for a DATASOURCE object the Connect method is never called then the default connection of the report engine will be assumed. Therefore, if you are passing the data access connection dynamically from the host application to the report engine, and you are using only one such connection throughout your reports, then you do not need to call the Connect method for your DATASOURCE objects in the reports.

ELS-Row and ResultRow

Recall that in the "**A Simple Example**" section of this chapter, we have outlined the script of a simple report, which utilizes a tabular presentation of data via direct HTML tables. Moreover, HTML table defining the tabular presentation was intertwined directly with the logic of the iteration over the data source. This intertwining may sometimes be a little undesirable, especially when the same tabular format is to be used at several different points

in the report generation process. A few other somewhat undesirable aspects of HTML tables are the automatic wrapping, as well as, the rigid grid structure. For example, it is not easy to make two rows in a single HTML table to have completely independent column structure. Of course, there are such attributes as COLSPAN, with which one may try to obtain different column numbers and widths for two rows, but this concept is based on grid-like structure of the HTML table, and moreover, it may get quite complicated to code, as is illustrated in the following HTML code example:

```
<table border="0" cellpadding="0" cellspacing="0" width="100%">
    <tr>
            <td width="24%" colspan="2">Col11</td>
            <td width="12%">Col12</td>
            <td width="38%" colspan="3">Col13</td>
            <td width="13%">Col14</td>
            <td width="13%">Col15</td>
    </tr>
    <tr>
            <td width="12%">Col21</td>
            <td width="24%" colspan="2">Col22</td>
            <td width="12%">Col23</td>
            <td width="39%" colspan="3">Col24</td>
            <td width="13%">Col25</td>
    </tr>
    <tr>
            <td width="12%">Col31</td>
            <td width="12%">Col32</td>
            <td width="37%" colspan="3">Col33</td>
            <td width="13%">Col34</td>
            <td width="13%">Col35</td>
            <td width="13%">Col36</td>
    </tr>
</table>
```

Because of these shortcomings of the HTML tables, the SCRIPT language has a special tabulation element called *ELS-Row*. This element is defined in the same spirit as HTML tables, namely via tags and their respective attributes. The following list defines all the tags that together compose the *ELS-Row* element:

<ELS_ROW>, </ELS_ROW>	begin and end tags of the *ELS-Row* element,
<L>, </L>	begin and end tags of the *ELS-Line* element,
<C>, </C>	begin and end tags of the *ELS-Column* element,
<HDR>, </HDR>	begin and end tags of the column header element,
<FLD>, </FLD>	begin and end tags of the field element.

An *ELS-Row* element consists of one or more *ELS-Line* elements. An *ELS-Line* element consists of one or more *ELS-Column* elements. The *ELS-Column* elements consist of *HDR* or *FLD* elements, intertwined with HTML segments. Therefore, the general syntax for an *ELS-Row* construct may be as follows:

```
<ELS_ROW NAME=els_row_spec_name els_row_attributes>
    <L els_line_attributes_1>
            <C els_column_attributes_11>
            ...HDR or FLD elements intertwined with HTML segments...
            </C>
            ...
            <C els_column_attributes_1M1>
            ...HDR or FLD elements intertwined with HTML segments...
            </C>
    </L>
    <L els_line_attributes_2>
            <C els_column_attributes_21>
            ...HDR or FLD elements intertwined with HTML segments...
            </C>
            ...
            <C els_column_attributes_2M2>
```

```
            ...HDR or FLD elements intertwined with HTML segments...
            </C>
    </L>
    ...
    <L els_line_attributes_N>
            <C els column attributes N1>
            ...HDR or FLD elements intertwined with HTML segments...
            </C>
            ...
            <C els_column_attributes_NM_N>
            ...HDR or FLD elements intertwined with HTML segments...
            </C>
    </L>
</ELS_ROW>
```

where *els_row_spec_name* is a unique name string identifying the *ELS-Row* tabulation specification, the *els_row_attributes* is a list of other *ELS-Row* attributes, the *els_line_attributes_i* are lists of *ELS-Line* attributes for respective *i*-th *ELS-Line* elements, and the *els_column_attributes_ij* are lists of *ELS-Column* attributes for respective *ij*-th *ELS-Column* elements.

Note that in the general syntax of *ELS-Row* element, the attributes and the number of columns from one *ELS-Line* to another may vary in a completely arbitrary manner, and therefore compared to HTML tables, *ELS-Rows* have a much more flexible tabulation mechanism. But before we begin the excursion course into the realms of complex tabulations of datasets via *ELS-Row* elements, we need to outline the rules and attributes of *ELS-Row* elements.

Each <ELS_ROW> tag must be uniquely identified via the mandatory NAME attribute value. We will find out in a little while, why the specification of this NAME attribute is mandatory. All attribute values must be specified in between double-quotes, single quotes or missing quotes are considered invalid values and will generate a "meaningless text" error when the script is compiled.

The complete listing of attributes of the <ELS_ROW> tag is outline in the following table:

Attribute	Description
NAME	The value of this attribute must be unique string identifying the *ELS-Row* element. The NAME attribute is mandatory, and therefore must be specified for all *ELS-Row* elements.
ALIGN	This attribute controls the *ELS-Row* element's position relative to the HTML body or the parent element. It can assume only the following values (in double-quotes): left (default), center and right
BACKGROUND	This attribute controls the background image of the *ELS-Row* element. By default the background of an *ELS-Row* element is transparent.
BGCOLOR	This attribute controls the background color of the *ELS-Row* element, by default the background is transparent. Valid values are either named colors, such as red, blue, etc. or the RGB values in hex codes with the following format: #hhhhhh
BORDER	This attribute controls the exterior border of the tabulation resulting from the *ESL-Row* element. By default the value is 0. Valid values may be specified in pixels, points, inches or any other unit (e.g. "90px", "10.8pt", "1.5in", etc.)
BORDERCOLOR	This attribute controls the border color of all cells and the exterior border as well. By default this value is empty (i.e. transparent). Valid values are either named colors, such as red, blue, etc. or the RGB values in hex codes with the following format: #hhhhhh
BORDERCOLORDARK	This attribute controls the border color of cells and the exterior frame which lies on the shade side of the frame. Valid values are either named colors, such as red, blue, etc. or the RGB values in hex codes with the following format: #hhhhhh
BORDERCOLORLIGHT	This attribute controls the border color of cells and the exterior frame which lies on the light-source side of the frame. Valid values are either named colors, such as red, blue, etc. or the RGB values in hex codes with the following format: #hhhhhh
CELLPADDING	This attribute controls the padding within cells (i.e. *ELS-Column* elements). By default the value is 0. Valid values may be specified in pixels, points, inches or any other unit.
CELLSPACING	This attribute controls the space between cells (i.e. *ELS-Column* elements). By default the value is 0. Valid values may be specified in pixels, points, inches or any other unit.

CLASS	This attribute specifies style sheet classes that the element belongs to. Values may be a space separated list of style sheet class names.
DIR	This attribute controls the directionality of text. Valid values are "ltr" and "rtl". By default the value is "ltr".
FRAME	This attribute specifies which sides of the frame surrounding the *ELS-Row* will be visible. Possible values are: void, above, below, hsides, lhs, rhs, vsides, box and border.
HEIGHT	This attribute controls the height of the *ESL-Row* element. Valid values may be specified in pixels, points, inches or any other unit, or in percents (e.g. "90px", "10.8pt", "1.5in", "75%", etc.).
ID	This attribute identifies the element. Not very useful in SCRIPT language.
LANG	This attribute specifies the language code.
LANGUAGE	This attribute specifies the predefined script language name.
RULES	This attribute controls the rulings between rows and columns. Possible values are: none, groups, rows, cols and all.
STYLE	This attribute controls the style properties of the *ELS-Row* element, and it is the recommended method of controlling the format behavior of elements in SCRIPT language.
WIDTH	This attribute controls the width of the *ESL-Row* element. Valid values may be specified in pixels, points, inches or any other unit, or in percents (e.g. "90px", "10.8pt", "1.5in", "75%", etc.)

Table 2.2. *The attributes of ELS_ROW tag*

Before we continue with our outline of the attributes of *ELS-Line* and *ELS-Column* elements, we should emphasize the use of STYLE attribute in <ELS_ROW> tags. Ever since the introduction and the stabilization of the *Cascaded Style Sheet*, as well as, the *XHTML* standards, the use of style sheets has immerged to become a more preferred method to specify properties of HTML tags or elements. Following in this spirit, we first list all property names of the STYLE attribute that are support by the IE5.5 browser.

ACCELERATOR	COLOR	OVERFLOW-Y	TEXT-AUTOSPACE
BACKGROUND-ATTACHMENT	CURSOR	PADDING-BOTTOM	TEXT-DECORATION
BACKGROUND-COLOR	DIRECTION	PADDING-LEFT	TEXT-DECORATION-BLINK
BACKGROUND-IMAGE	DISPLAY	PADDING-RIGHT	TEXT-DECORATION-LINETHROUGH
BACKGROUND-POSITION-X	FILTER	PADDING-TOP	TEXT-DECORATION-NONE
BACKGROUND-POSITION-Y	FLOAT	PAGE-BREAK-AFTER	TEXT-DECORATION-OVERLINE
BACKGROUND-REPEAT	FONT-FAMILY	PAGE-BREAK-BEFORE	TEXT-DECORATION-UNDERLINE
BEHAVIOR	FONT-SIZE	POS-BOTTOM	TEXT-INDENT
BORDER-BOTTOM	FONT-STYLE	POS-HEIGHT	TEXT-JUSTIFY
BORDER-BOTTOM-COLOR	FONT-VARIANT	POSITION	TEXT-JUSTIFY-TRIM
BORDER-BOTTOM-STYLE	FONT-WEIGHT	POS-LEFT	TEXT-KASHIDA
BORDER-BOTTOM-WIDTH	HEIGHT	POS-RIGHT	TEXT-KASHIDA-SPACE
BORDER-COLLAPSE	IME-MODE	POS-TOP	TEXT-TRANSFORM
BORDER-COLOR	LAYOUT-FLOW	POS-WIDTH	TEXT-UNDERLINE-POSITION
BORDER-LEFT	LAYOUT-GRIDCHAR	RIGHT	TOP
BORDER-LEFT-COLOR	LAYOUT-GRIDLINE	RUBY-ALIGN	UNICODE-BIDI
BORDER-LEFT-STYLE	LAYOUT-GRIDMODE	RUBY-OVERHANG	VERTICAL-ALIGN
BORDER-LEFT-WIDTH	LAYOUT-GRIDTYPE	RUBY-POSITION	VISIBILITY
BORDER-RIGHT	LEFT	SCROLLBAR-3DLIGHT-COLOR	WHITE-SPACE
BORDER-RIGHT-COLOR	LETTER-SPACING	SCROLLBAR-ARROW-COLOR	WIDTH
BORDER-RIGHT-STYLE	LINE-BREAK	SCROLLBAR-BASE-COLOR	WORD-BREAK
BORDER-RIGHT-WIDTH	LINE-HEIGHT	SCROLLBAR-DARKSHADOW-COLOR	WORD-SPACING
BORDER-STYLE	LIST-STYLE-IMAGE	SCROLLBAR-FACE-COLOR	WORD-WRAP
BORDER-TOP	LIST-STYLE-POSITION	SCROLLBAR-HIGHLIGHT-COLOR	WRITING-MODE
BORDER-TOP-COLOR	LIST-STYLE-TYPE	SCROLLBAR-SHADOW-COLOR	Z-INDEX
BORDER-TOP-STYLE	MARGIN-BOTTOM	SCROLLBAR-TRACK-COLOR	ZOOM
BORDER-TOP-WIDTH	MARGIN-LEFT	TABLE-LAYOUT	
BORDER-WIDTH	MARGIN-RIGHT	TEXT-ALIGN	
BOTTOM	MARGIN-TOP	TEXT-ALIGN-LAST	
CLEAR	OVERFLOW		
CLIP	OVERFLOW-X		

Table 2.3. *All possible property names of the* STYLE *attribute*

Properties of STYLE attribute are specified in a semicolon separated string, with each property stated in the form

name:*value*, for example:

```
style="FONT-SIZE:32pt; BACKGROUND-COLOR:blue; TEXT-ALIGN:center; OVERFLOW:hidden"
```

where *value* is an appropriate valid value depending on the name type. To get the reader acquainted with the value types of properties of STYLE attribute, we advice checking the **Properties** pane in **Design** view. In general, the value types fall in one of the following categories: *color*, *uri* (e.g. *url*), *percentage*, *length*, *string*, *identifier*, *integer*, *size*, *number*, *width* and *time*.

We are ready now to continue the attribute listing of the *ELS-Row*. In particular, we consider next the attributes of the <L> tags (see Table 2.4):

Attribute	Description
ALIGN	This attribute specifies the alignment of data and the justification of text in a cell (i.e. *ELS-Column* element). It can assume only the following values (in double-quotes): left (default), center, right and justify.
BACKGROUND	This attribute controls the background image of the *ELS-Line* element. By default the background of an *ELS-Line* element is transparent.
BGCOLOR	This attribute controls the background color of the *ELS-Line* element, by default the background is transparent. Valid values are either named colors, such as red, blue, etc. or the RGB values in hex codes with the following format: #hhhhhh
BORDER	This attribute controls the exterior border of the tabulation resulting from the *ESL-Line* element. By default the value is 0. Valid values may be specified in pixels, points, inches or any other unit (e.g. "90px", "10.8pt", "1.5in", etc.)
BORDERCOLOR	This attribute controls the border color of all cells and the exterior border as well. By default this value is empty (i.e. transparent). Valid values are either named colors, such as red, blue, etc. or the RGB values in hex codes with the following format: #hhhhhh
BORDERCOLORDARK	This attribute controls the border color of cells and the exterior frame which lies on the shade side of the frame. Valid values are either named colors, such as red, blue, etc. or the RGB values in hex codes with the following format: #hhhhhh
BORDERCOLORLIGHT	This attribute controls the border color of cells and the exterior frame which lies on the light-source side of the frame. Valid values are either named colors, such as red, blue, etc. or the RGB values in hex codes with the following format: #hhhhhh
CELLPADDING	This attribute controls the padding within cells (i.e. *ELS-Column* elements). By default the value is 0. Valid values may be specified in pixels, points, inches or any other unit.
CELLSPACING	This attribute controls the space between cells (i.e. *ELS-Column* elements). By default the value is 0. Valid values may be specified in pixels, points, inches or any other unit.
CLASS	This attribute specifies style sheet classes that the element belongs to. Values may be a space separated list of style sheet class names.
DIR	This attribute controls the directionality of text. Valid values are "ltr" and "rtl". By default the value is "ltr".
FRAME	This attribute specifies which sides of the frame surrounding the *ELS-Line* will be visible. Possible values are: void, above, below, hsides, lhs, rhs, vsides, box and border.
HEIGHT	This attribute controls the height of the *ESL-Line* element. Valid values may be specified in pixels, points, inches or any other unit, or in percents (e.g. "90px", "10.8pt", "1.5in", "75%", etc.).
ID	This attribute identifies the element. Not very useful in SCRIPT language.
LANG	This attribute specifies the language code.
LANGUAGE	This attribute specifies the predefined script language name.
RULES	This attribute controls the rulings between rows and columns. Possible values are: none, groups, rows, cols and all.
STYLE	This attribute controls the style properties of the *ELS-Line* element, and it is the recommended method of controlling the format behavior of elements in SCRIPT language.

VALIGN	This attribute specifies the vertical position of data within cell (i.e. *ELS-Column* element). Possible values are top, middle, bottom and baseline.
WIDTH	This attribute controls the width of the *ESL-Line* element. Valid values may be specified in pixels, points, inches or any other unit, or in percents (e.g. "90px", "10.8pt", "1.5in", "75%", etc.)

Table 2.4. *All possible attributes of the ELS-Line elements*

There are very little differences between attributes of the *ELS-Row* and that of the *ELS-Line*. In contrast, the attributes of the *ELS-Column* have some significant differences in terms of interpretation. The following table lists all the relevant attributes of the *ELS-Column* elements:

Attribute	Description
ABBR	This attribute should be used to provide abbreviated form of the cell's (i.e. *ELS-Column*'s) content.
ALIGN	This attribute specifies the alignment of data and the justification of text in a cell (i.e. *ELS-Column* element). It can assume only the following values (in double-quotes): left (default), center, right and justify.
BACKGROUND	This attribute controls the background image of the *ELS-Column* element. By default the background of an *ELS-Column* element is transparent.
BGCOLOR	This attribute controls the background color of the *ELS-Column*, by default the background is transparent. Valid values are either named colors, such as red, blue, etc. or the RGB values in hex codes with the following format: #hhhhhh
BORDERCOLOR	This attribute controls the border color of the *ELS-Column*. Valid values are either named colors, such as red, blue, etc. or the RGB values in hex codes with the following format: #hhhhhh
BORDERCOLORDARK	This attribute controls the segment of the border color of the *ELS-Column* that lies on the shade side. Valid values are either named colors, such as red, blue, etc. or the RGB values in hex codes with the following format: #hhhhhh
BORDERCOLORLIGHT	This attribute controls the segment of the border color of the *ELS-Column* that lies on the light source side. Valid values are either named colors, such as red, blue, etc. or the RGB values in hex codes with the following format: #hhhhhh
CLASS	This attribute specifies style sheet classes that the element belongs to. Values may be a space separated list of style sheet class names.
COLSPAN	This attribute specifies the number of columns the cell spans over. Not very useful when used in *ELS-Column* elements.
DIR	This attribute controls the directionality of text. Valid values are "ltr" and "rtl". By default the value is "ltr".
HEIGHT	This attribute controls the height of the *ESL-Column* element. Valid values may be specified in pixels, points, inches or any other unit, or in percents (e.g. "90px", "10.8pt", "1.5in", "75%", etc.).
ID	This attribute identifies the element. Not very useful in SCRIPT language.
LANG	This attribute specifies the language code.
LANGUAGE	This attribute specifies the predefined script language name.
NOWRAP	This attribute specifies to disable automatic text wrapping for the *ELS-Column* element.
ROWSPAN	This attribute specifies the number of rows the cell spans over.
STYLE	This attribute controls the style properties of the *ELS-Column* element, and it is the recommended method of controlling the format behavior of elements in SCRIPT language.
VALIGN	This attribute specifies the vertical position of data within cell (i.e. *ELS-Column* element). Possible values are top, middle, bottom and baseline.
WIDTH	This attribute controls the width of the *ESL-Column* element. Valid values may be specified in pixels, points, inches or any other unit, or in percents (e.g. "90px", "10.8pt", "1.5in", "75%", etc.)

Table 2.5. *All possible attributes of the ELS-Column elements*

So far we have covered all parts of an *ELS-Row* element up to the *ELS-Column* level. The *ELS-Column* elements are essentially the cells of the *ELS-Row*, and therefore may contain *HDR* or *FLD* elements intertwined with HTML segments. In addition, to the <ELS_ROW>, <L> and <C> tags, the SCRIPT language has another tag sometimes used in *ELS-Row* elements. This tag is denoted by <ELS_SHAPE> and is primarily used to divide an *ELS-Row* element into named subsections. We will describe more about the *ELS-Shape* elements later on in this document, for now, we will avoid using *ELS-Shapes*.

The main purpose of the *ELS-Row* element is to provide a row specification for the data presentation in the report output. Moreover, this row specification is identified by the unique name value of the NAME attribute. The data flow, on the other hand, is controlled via the *ResultRow* or *ResultRows* functions, which we define next. The *ResultRow* function has the following syntax:

```
ResultRow(VARCHAR sRowSpecName [, VARCHAR sDSSpecName ] );
```

where the second argument is optional (as indicated by the square brackets) and will be by default NULL if not specified. This function applies the row specification referenced by the sRowSpecName over the current record of the data set (i.e. a single record). The *ResultRows* function has the following syntax:

```
ResultRows(DATASOURCE oDataSource, VARCHAR sRowSpecName [, VARCHAR sDSSpecName ] );
```

where oDataSource is the data source on which the row specification defined by sRowSpecName spans, while the third argument is optional *ELS-Shape* row specification. The difference between the *ResultRow* and *ResultRows* functions is that *ResultRow* applies only over the current record, whereas *ResultRows* applies over the whole data set defined by the first argument. We illustrate the use of these functions in the following code snippet:

```
// ELS-Row "ELSRowMain" must be specified before this line
// with ELS-Shapes "Orders" and "OrderDetails"
<ELS>
    // set the parent recordset @ds
    SET @ds = @sSQL;

    WHILE NOT @ds.EOF()
           // display the parent records and set the child recordset
           ResultRow("ELSRowMain", "Orders");
           SET @dsChild = @ds.CHILD("OrdID");
           // display all Order Details records for the current OrderID
           ResultRows(@dsChild, "ELSRowMain", "OrderDetails");
           @ds.NEXT();
    END LOOP
</ELS>
```

Note that in this code snippet we have used data shape recordsets, and linked these recordsets via the *CHILD* method of the parent recordset. We should point out that this code snippet is not at all complete, and should only serve for the purpose of illustrating the syntax usage of the *ResultRow* and *ResultRows* functions.

In this regards, we should emphasize also that the data presentation location in the report output is essentially determined by the location of the *ResultRow* or *ResultRows* function calls, while the *ELS-Row* specification only defines the format and structure of the tabulation.

Recall that a *FLD* element, defined by the <FLD> and </FLD> tags, is essentially used to put the result of any SCRIPT expression or field into the report output as HTML segments. It can be utilized only outside pure *ELS* sections, while its content can be pure *ELS* elements. Recall also, that pure *ELS* elements consist of valid SCRIPT expressions or data fields, and that the main function of the <FLD> and </FLD> tags, is to convert the run-time value of the specified SCRIPT expression to character string representation to be put as HTML segment into the report output. A *HDR* element, on the other hand, is defined via <HDR> and </HDR> tags, and may contain *HTML*, as well as, *FLD* elements. The *HDR* elements control the column headers of the tabulation used in a report, and have special functions that control the repetition patterns of these headers in nested row sets.

To illustrate the usage of the *FLD* and *HDR* elements, we first convert the simple example that we outlined in previous sections to a form that utilizes *ELS-Row* elements. To proceed in this direction, you may use the **Project**

Explorer pane's popup menu, to copy and paste a copy of the existing Report1.ELS report file into the project. Then rename this copy as Report2.ELS and open it into the SCRIPT *Editor* window. The first task that we must perform is to remove the content of the *ELS_RDETAIL* report section. To do this, select the text in between the <ELS_RDETAIL> and </ELS_RDETAIL> tags and delete it from the script.

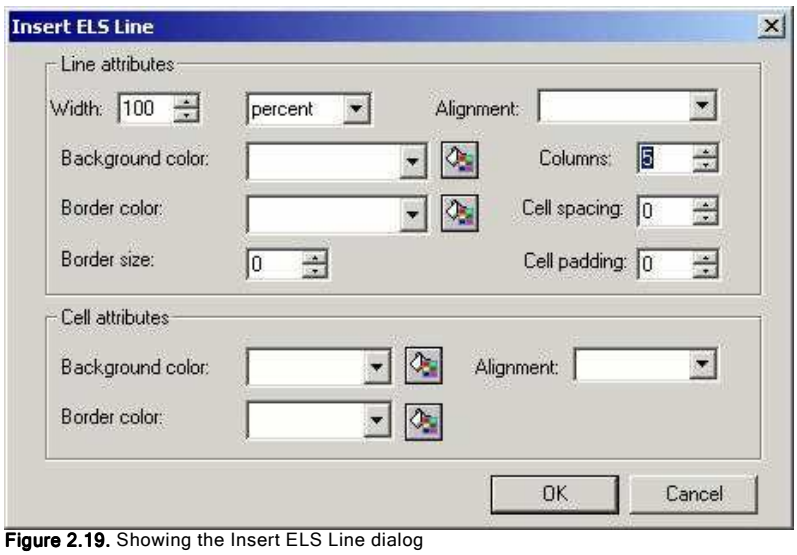


Figure 2.19. Showing the Insert ELS Line dialog

The next task will be to add an *ELS-Row* element inside the *ELS_RDETAIL* section. To accomplish this, put the cursor between <ELS_RDETAIL> and </ELS_RDETAIL> tags and call the **Insert ELS-Row** resizable dialog via the **ELS Row** menu item of the **Insert** menu (or alternatively via the **Insert ELS Row** toolbar button). In the **Insert ELS-Row** dialog, uncheck the **Only ELS-Row Spec** check-box, so that the **Datasource** combo-box becomes enabled. Drop down the **Datasource** combo-box and select the @MyOrders from the existing data sources defined in the report script. In general, this combo-box will contain all the data source variable names declared in the current report script. Observe that the **Name** edit-box already contains a suggested *ELS-Row* element name, for example ELSRow1. Next, note the toolbar buttons in this **Insert ELS-Row** dialog, and in particular, click on the **Add Line** button, which is the second button from left. This will prompt the **Insert ELS Line** dialog shown in Figure 2.19. In this dialog make sure the **Width** is 100 percent, set the **Border size** to 0 pixels, and increase the **Columns** to 5. Submitting this **Insert ELS Line** dialog will insert a line into the client area of the **Insert ELS-Row** window (see Figure 2.20 for more details). The user may add more *ELS-Lines* or move selected line up or down, as well as, delete or duplicate such lines via the toolbar buttons. Once an *ELS-Line* is inserted into the client area of the **Insert ELS-Row** window, the user can resize the columns of such line via mouse-button press and move combination. Observe in Figure 2.20 that we have resized the column widths to proper sizes to fit the requirements of the simple example.

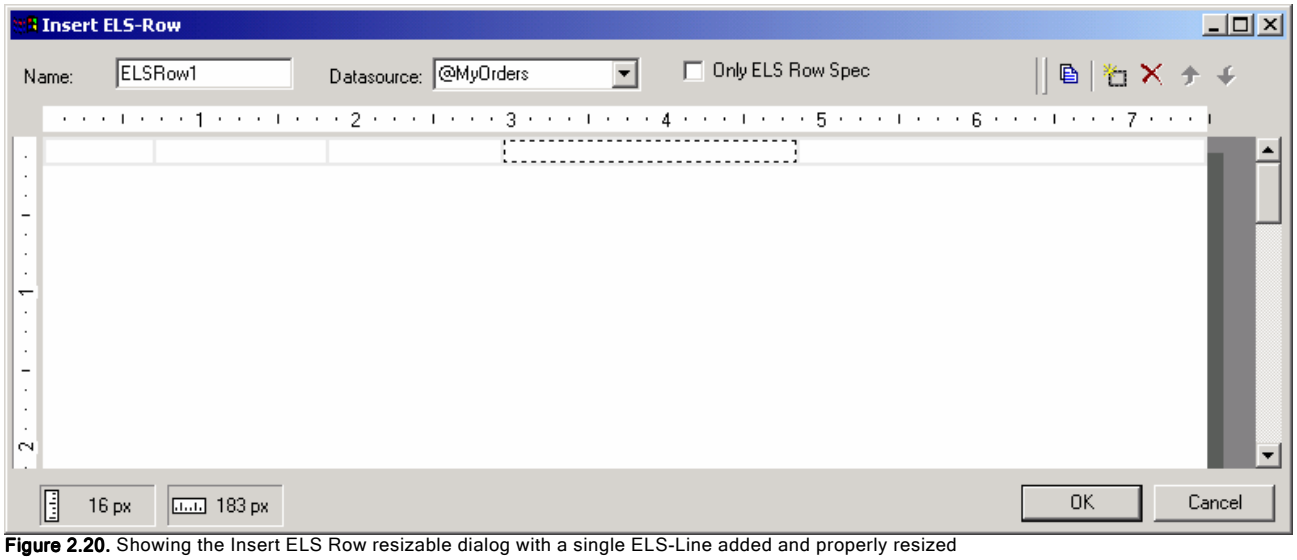


Figure 2.20. Showing the Insert ELS Row resizable dialog with a single ELS-Line added and properly resized

When the user submits this **Insert ELS-Row** window, the following empty *ELS-Row* element will be inserted at the cursor location in the *ELS_RDETAIL* section, together with the iteration control logic to generate the data presentation:

```
<ELS RDETAIL FONT-FAMILY="Times New Roman" FONT-SIZE="9pt">
<ELS_ROW NAME="ELSRow1">
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
            <C WIDTH="8%" HEIGHT="15">
                     
            </C>
            <C WIDTH="12%" HEIGHT="15">
                     
            </C>
            <C WIDTH="12%" HEIGHT="15">
                     
            </C>
            <C WIDTH="28%" HEIGHT="15">
                     
            </C>
            <C WIDTH="40%" HEIGHT="15">
                     
            </C>
    </L>
</ELS ROW>
<ELS>
WHILE NOT @MyOrders.Eof()
    ResultRow("ELSRow1");
    @MyOrders.Next();
END LOOP
</ELS>

</ELS RDETAIL>
```

Observe that by default each <C> tag contains only the HTML code for the space character, namely " ". We will replace these empty cells with the desired data fields utilizing the **Data Fields** window, but before we do this we advise the user on couple of relevant points. The first is to change the default format option of date-time data types to short dates. This can be performed via the **Options** dialog's **Data** tab. In this tab, click on the button corresponding to the **Date Time** data type, and in **Format/Conversion Wizard** dialog change the **Format** combo-box value to Short Date, and then submit both dialogs.

Now, using the **Data Fields** window, insert the desired fields via drag-drop into the lines that contain the " ". Also, we manually enter the desired column header caption wrapped between <HDR> and </HDR> tags, so that *ELS_RDETAIL* report section becomes as follows:

```
<ELS_RDETAIL FONT-FAMILY="Times New Roman" FONT-SIZE="9pt">
<ELS ROW NAME="ELSRow1">
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
            <C WIDTH="8%" HEIGHT="15">
            <HDR><B>OrderID</B></HDR>           // column header for column 1
             <FLD>Format(@MyOrders.Column("OrderID"), "")</FLD>
            </C>
            <C WIDTH="12%" HEIGHT="15">
            <HDR><B>Order Date</B></HDR>        // column header for column 2
             <FLD>Format(@MyOrders.Column("OrderDate"), "mm/dd/yyyy")</FLD>
            </C>
            <C WIDTH="12%" HEIGHT="15">
            <HDR><B>Ship Date</B></HDR>         // column header for column 3
             <FLD>Format(@MyOrders.Column("ShippedDate"), "mm/dd/yyyy")</FLD>
            </C>
            <C WIDTH="28%" HEIGHT="15">
            <HDR><B>Ship Name</B></HDR>         // column header for column 4
             <FLD>@MyOrders.Column("ShipName")</FLD>
            </C>
            <C WIDTH="40%" HEIGHT="15">
            <HDR><B>Ship Address</B></HDR>      // column header for column 5
             <FLD>@MyOrders.Column("ShipAddress")</FLD>
```

```
            </C>
    </L>
</ELS_ROW>
<ELS>
BeginHeader("ELSRow1");
WHILE NOT @MyOrders.Eof()
    ResultRow("ELSRow1");
    @MyOrders.Next();
END LOOP
EndHeader("ELSRow1");
</ELS>

</ELS_RDETAIL>
```

Note that when inserting fields of date-time data type, the adjusted **Options** dialog's default format specification of "mm/dd/yyyy" was automatically used in the data field insertion. One other crucial point concerning the advantages of using *ELS-Row* elements compared to HTML tables for tabulation of data presentation, is that in *ELS-Row* elements you may specify column widths as attributes to the <C> tags with percents rather than using <SPAN> tags with unit measures. Moreover, the SCRIPT engine has internal mechanisms to handle column headers when using *HDR* elements inside *ELS-Row* elements combined with proper *HDR* function calls. For example, to make column headers appear at the beginning of each page of the report detail section, all one has to do is to include *HDR* elements in the *ELS-Row* specification, as we did in the previous code segment, and utilize the *BeginHeader* and *EndHeader HDR* functions (to be described shortly). As a result of this later fact, we should remove the old code from the *OnBeginPage* event that used to handle the column headers in the HTML table case. This completes the simple example, with the HTML table replaced with a much more powerful *ELS-Row* element.

We now are ready to describe the *HDR* functions in more details. Essentially, these functions control the pattern of the column headers used in *ELS-Row* elements. To begin the discussion of this topic, it is helpful to view the column headers as a collection of header rows, with a well-defined order. And that this collection is repeated at the beginning of each page, as well as, at the beginning of each group subsection of the report. We list these column-headers pattern manipulation functions next:

`BeginHeader(sRowName [, sShapeName])`	this function first writes the header row specified by the *ELS-Row* `sRowName` (and optionally by *ELS-Shape* `sShapeName` subsection of the *ELS-Row*), and then it adds this header row to the internal column-headers pattern collection object.
`EndHeader(sRowName [, sShapeName])`	this function marks off the header row specified by the *ELS-Row* `sRowName` (and optionally by *ELS-Shape* `sShapeName` subsection of the *ELS-Row*) from the internal column-headers pattern collection object.

We summarize next the rules involved in the manipulation of column-headers pattern. In particular, header rows must behave according to the following rules:

1. There are several ways to define header rows in an *ELS-Row* specification, we may include both *HDR* and *FLD* elements in the same <C> tags of a single <L> tag, or in the <C> tags of two or more different <L> tags. The second method is desirable especially when the background properties of the *HDR* and *FLD* elements need to be different or their column structure more independent.
2. We may include *FLD* elements along with HTML elements inside *HDR* elements.
3. A header row may span over more than a single <L> line, just as the field or data rows do.
4. A header row consisting of multiple <L> lines, behaves as a single indivisible unit with respect to page mechanisms. Therefore such a header row will not be split across pages.
5. To implement column-header lines that do split across pages, the user may use several header rows each consisting of a single <L> line.
6. The writing of the header rows in the report output are controlled via explicit use of *BeginHeader* and *EndHeader* functions.
7. By default, a header row is repeated automatically at the beginning of the subsequent pages after the corresponding *BeginHeader* call, provided the corresponding *EndHeader* function is not called prior to the occurrence of the new page event.
8. An *ELS-Row* that contains only *HDR* or HTML elements is called a *pure header row specification*. For

such a row specification calling *ResultRow* or *ResultRows* will have no effect on the report output. A *pure header row specification* may be used to associate a header row with a data row with completely different column structure.

9. An *ELS-Row* that contains no *HDR* elements is called a *header-less row specification*. For such a row specification calling *BeginHeader* will have no effect on the report output.

We consider next a report consisting of a parent data set, with each record of this parent data set possessing a detailed child data set. For this example we use a data shape consisting of `Orders` and `[Order Details]` tables. To learn how to build data shape commands we invite the user to read the sections in Chapter 1 relating to data shapes. Therefore, for the current example, we may assume that a data shape object corresponding to `Orders` and `[Order Details]` exists in the **Datashapes** child node of the **AUX** node of the **Project Explorer** pane. From the **Project Explorer** pane use the popup menu commands to copy and paste a copy of the Report2.ELS file into the project (assuming that MyProj1 report project is open in the *Report Designer* application). Then rename this new report file as Report3.ELS, and open it in the SCRIPT *Editor* window. The first thing we will do is to remove the `@MyOrders` data source declaration and definition (i.e. the `SET` statement). Also, empty the *ELS-RDETAIL* report section so that we can start from fresh.

In the **Project Explorer** pane, expand the **MyNWDB** connection node under the **Databases** node, then expand the child node **AUX**, finally expand the **Datashapes** child node of the **AUX** node. Recall that we have assumed that a data shape, say OrdersShp was already created. Therefore, drag-drop the existing OrdersShp data shape into the script, at the end of the section defined by the `<ELS_RSETTINGS>` and `</ELS_RSETTINGS>` tags. This will result to the following code:

```
<ELS_RSETTINGS>
    SET REPORT_TITLE      = "Report Title";
    SET PAGE_ORIENTATION  = ELS_PORTRAIT;
    SET PAGE_SIZE         = ELS_LETTER;
    SET PAGE_SOURCE       = ELS_UPPER;
    SET PAGE_MARGINS.LEFT        = 0.5;
    SET PAGE_MARGINS.RIGHT       = 0.5;
    SET PAGE_MARGINS.TOP         = 0.75;
    SET PAGE_MARGINS.BOTTOM      = 0.5;
    SET DEFAULTMODE              = ELS_FAST;
    SET SUPPRESS_PHEADER.FIRSTPAGE = TRUE;
    SET SUPPRESS_PFOOTER.FIRSTPAGE = TRUE;

    DECLARE @OrdersShp DATASOURCE;
    SET @OrdersShp = "SHAPE " +
                         "{SELECT " +
                                "* " +
                         "FROM " +
                                "Orders} AS Orders " +
                  "APPEND " +
                  "( " +
                         "{SELECT " +
                                "* " +
                         "FROM " +
                         "[Order Details] } AS OrdDetails " +
                  "RELATE " +
                         "OrderID TO OrderID " +
                  ") AS OrdDetails";

    // this data source variable is needed for the child column of the data shape
    DECLARE @OrdDetails DATASOURCE;

</ELS_RSETTINGS>
```

Also note that we have declared a new `DATASOURCE` variable `@OrdDetails` to be used for the child column of the data shape command.

In our next step, we like to define a parent row consisting of multiple lines on which fields *OrderID*, *OrderDate*, *RequiredDate*, *ShippedDate*, *ShipVia*, *Freight*, as well as, shipping address information are arranged with field names (indicated in bold text), together with the corresponding data values (indicated by grey background):

ORDER ID:	10248		SHIPPING INFORMATION:
ORDER DATE:	07/04/1996		Vins et alcools Chevalier
REQUIRED DATE:	08/01/1996		59 rue de l'Abbaye
SHIPPED DATE	SHIP VIA	FREIGHT	Reims, CA_X 51100
07/16/1996	3	$32.38	France

Observe that such a tabular presentation may be achieved via an *ELS-Row* consisting of three *ELS-Lines* of 3 columns and two *ELS-Lines* of 4 columns. Therefore, call the **Insert ELS-Row** window via the **ELS Row** menu command of the **Insert** menu, and in this window first add an *ELS-Line* with 3 columns. After resizing columns properly, duplicate this *ELS-Line* twice to form the three desired lines. Then add an *ELS-Line* with 4 columns, resize properly and then duplicate to complete the whole *ELS-Row* structure. When all the lines are arranged and the **Insert ELS-Row** window submitted, the following script will be inserted at the cursor location in the *ELS_RDETAIL* report section:

```
<ELS_ROW NAME="ELSRow1">
    <L border="0" CellSpacing="0" CellPadding = "0" WIDTH="100%" HEIGHT="15">
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                   
           </C>
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="27%">
                   
           </C>
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="56%">
                   
           </C>
    </L>
    <L border="0" CellSpacing="0" CellPadding = "0" WIDTH="100%" HEIGHT="15">
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                   
           </C>
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="27%">
                   
           </C>
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="56%">
                   
           </C>
    </L>
    <L border="0" CellSpacing="0" CellPadding = "0" WIDTH="100%" HEIGHT="15">
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                   
           </C>
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="27%">
                   
           </C>
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="56%">
                   
           </C>
    </L>
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                   
           </C>
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="10%">
                   
           </C>
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                   
           </C>
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="56%">
                   
           </C>
    </L>
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                   
           </C>
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="10%">
                   
           </C>
           <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
```



```
                 
            </C>
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="56%">
                 
            </C>
    </L>
</ELS_ROW>
```

Note that, the **Insert ELS-Row** window did not insert *ResultRow* calls since we did not uncheck the **Only ELS-Row Spec** checkbox control, and therefore as a result it only inserted *ELS-Row* specification.

Next we type in the field labels wherever applicable and insert the desired fields from the @OrdersShp data source via the **Data Fields** window. So that for example the script of the parent row may look like the following:

```
<ELS ROW NAME="ELSRow1">
    <L border="0" CellSpacing="0" CellPadding = "0" WIDTH="100%" HEIGHT="15">
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                <B>ORDER ID:</B>
            </C>
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="27%">
                <FLD>Format(@OrdersShp.Column("OrderID"), "")</FLD>
            </C>
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="56%">
                 <B>SHIPPING INFORMATION:</B>
            </C>
    </L>
    <L border="0" CellSpacing="0" CellPadding = "0" WIDTH="100%" HEIGHT="15">
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                <B>ORDER DATE:</B>
            </C>
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="27%">
                <FLD>Format(@OrdersShp.Column("OrderDate"), "mm/dd/yyyy")</FLD>
            </C>
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="56%">
                 <FLD>@OrdersShp.Column("ShipName")</FLD>
            </C>
    </L>
    <L border="0" CellSpacing="0" CellPadding = "0" WIDTH="100%" HEIGHT="15">
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                <B>REQUIRED DATE:</B>
            </C>
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="27%">
                <FLD>Format(@OrdersShp.Column("RequiredDate"), "mm/dd/yyyy")</FLD>
            </C>
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="56%">
                 <FLD>@OrdersShp.Column("ShipAddress")</FLD>
            </C>
    </L>
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                <B>SHIPPED DATE</B>
            </C>
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="10%">
                <B>SHIP VIA</B>
            </C>
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                <B>FREIGHT</B>
            </C>
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="56%">
                 <FLD>@OrdersShp.Column("ShipCity")</FLD>, 
                <FLD>@OrdersShp.Column("ShipRegion")</FLD> 
                <FLD>@OrdersShp.Column("ShipPostalCode")</FLD>
            </C>
    </L>
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                <FLD>Format(@OrdersShp.Column("ShippedDate"), "mm/dd/yyyy")</FLD>
            </C>
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="10%">
                <FLD>Format(@OrdersShp.Column("ShipVia"), "")</FLD>
            </C>
```

```
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="17%">
                  <FLD>Format(@OrdersShp.Column("Freight"), "$#,##0.00")</FLD>
            </C>
            <C BGCOLOR="palegoldenrod" HEIGHT="15" WIDTH="56%">
                   <FLD>@OrdersShp.Column("ShipCountry")</FLD>
            </C>
    </L>
</ELS_ROW>
```

Next, we apply the **Insert ELS-Row** again to insert a row specification for `[Order Details]` part of the data shape. This *ELS-Row* will consist of just a single *ELS-Line* and will display the following fields: *ProductID*, *UnitPrice*, *Quantity* and *Discount*. Since we also want an indentation at the beginning of each record, this *ELS-Row* must have 5 columns. Using the **Insert ELS-Row** window together with the **Data Fields** window, we insert the following script at a location after the "ELSRow1" row specification:

```
<ELS_ROW NAME="ELSRow2">
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
            <C bgColor="#FFFACD" WIDTH="20%" HEIGHT="15">
                  <HDR> </HDR> // this column is for indentation
            </C>
            <C bgColor="#FFFACD" WIDTH="20%" HEIGHT="15">
                  <HDR>ProductID</HDR>
            </C>
            <C bgColor="#FFFACD" WIDTH="20%" HEIGHT="15">
                  <HDR>Unit Price</HDR>
            </C>
            <C bgColor="#FFFACD" WIDTH="20%" HEIGHT="15">
                  <HDR>Quantity</HDR>
            </C>
            <C bgColor="#FFFACD" WIDTH="20%" HEIGHT="15">
                  <HDR>Discount</HDR>
            </C>
    </L>
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
            <C WIDTH="20%" HEIGHT="15">
                            // this column is for indentation
            </C>
            <C WIDTH="20%" HEIGHT="15">
                  <FLD>Format(@OrdDetails.Column("ProductID"), "")</FLD>
            </C>
            <C WIDTH="20%" HEIGHT="15">
                  <FLD>Format(@OrdDetails.Column("UnitPrice"), "$#,##0.00")</FLD>
            </C>
            <C WIDTH="20%" HEIGHT="15">
                  <FLD>Format(@OrdDetails.Column("Quantity"), "#,##0")</FLD>
            </C>
            <C WIDTH="20%" HEIGHT="15">
                  <FLD>Format(@OrdDetails.Column("Discount"), "$#,##0.00")</FLD>
            </C>
    </L>
</ELS_ROW>
```

Finally, we need to call the *HDR* and *ResultRow* functions inside prescribed iteration controls. The following SCRIPT code will iterate over all the records of `@OrdersShp` recordset, writing a parent record followed by iteration over the child recordset defined by `OrdDetails` child command of the data shape.

```
<ELS>

WHILE NOT @OrdersShp.Eof()
    ResultRow("ELSRow1");
    SET @OrdDetails = @OrdersShp.Child("OrdDetails");

    BeginHeader("ELSRow2");      // write the header row associated with ELSRow2
    WHILE NOT @OrdDetails.Eof()
            ResultRow("ELSRow2");
            @OrdDetails.Next();
    END LOOP
    EndHeader("ELSRow2");
```

```
    @OrdersShp.Next();
END LOOP

</ELS>
```

Note the use of the *BeginHeader* and *EndHeader* functions, which define the scope of the header row specified by "ELSRow2". So that whenever the iteration over the @OrdDetails recordset meets a new page, this header row will be automatically written at the beginning of the new page.

Using Auto-Align Wizard

In the previous section, we created a sample report script using data shape command as data source, presenting the information in the *Orders* and *Order Details* tables via a 5-lined *ELS-Row* element. In such situations, especially when several multilined *ELS-Row* elements are used, sometimes after some resize operations column alignments between *HDR*-lines and *FLD*-lines become out of synch. Moreover, utilizing resize operations in the **Design** view along with the **Precision Resizer** tool will get the columns approximately aligned, but sometimes we really need these alignment to be precise to the last pixel unit. For such purposes, we may use the **Auto-align Wizard**.

The **Auto-align Wizard** may be initiated in the **Source** view by placing the cursor anywhere between the <L> and </L> tags, or anywhere between the <TR> and </TR> tags, then using the right-mouse-button click method prompt the popup menu, and from this popup menu select the **Select for Auto-Align** menu command. This action will display the **Auto-align Wizard** modeless window with the <L> or <TR> element selected in the auto-alignment list-box. Subsequent repetion of this same action on different <L> or <TR> elements will simply add the element into the auto-alignment list-box provided these elements have the same number of columns as the topmost item in the auto-alignment list-box. For example, Figure 2.21 depicts the selection of the first three <L>-elements of the "ELSRow1" *ELS-Row* element in the sample report script of the previous section:

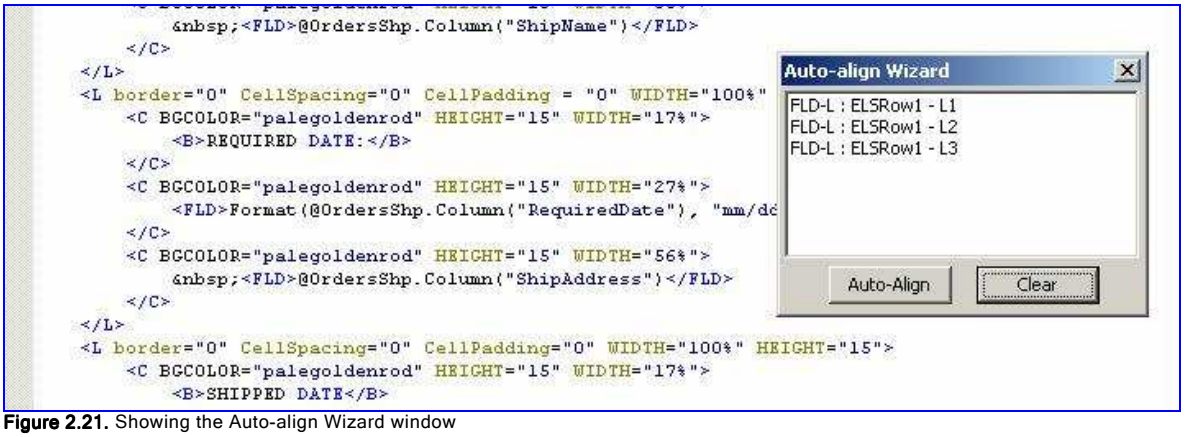


Figure 2.21. Showing the Auto-align Wizard window

Once the desired line elements are inserted into the auto-alignment list-box of the **Auto-align Wizard** window, the user may automatically align all the columns of these line elements by simply clicking the **Auto-Align** button. This action will make the columns of all lines in the list-box to have respectively the same column structure as that of the topmost item in the auto-alignment list-box. In other words, the topmost entry of the list-box of **Auto-align Wizard** window serves as a guide for all other line element entries in the list-box. Note that closing the dialog will clear this list and close the window, while the **Clear** button will empty this list-box.

Format and Conversion Function

We have used the *Format* function on several occasions in the sample codes that were used for the illustration of most of the concepts covered so far. It is about time to define this function in a more formal manner. This function is essentially used to convert data values from non-character data types into character types, and therefore has the following syntax:

```
VARCHAR Format(expression, format_spec)
```

where *expression* may be a valid SCRIPT expression of any data type, and *format_spec* is a character string specifying a valid format of the returned string. The rules for a valid format specification string depend on the data

type of the expression. In particular, this expression may have numeric, date-time or character data types, and for each of these categories the format specification has a separate set of rules. We begin by outlining the rules for date-time data types. For expressions that have date-time data types, the *format_spec* must be a combination of the following format characters:

Date-Time Format Spec	Description
d	Display the day as number with no leading zero (e.g. 1, 2, … , 31)
dd	Display the day as number with leading zero (e.g. 01, 02, … , 31)
ddd, DDD	Display the name of the day in abbreviated form (e.g. Sun, Mon, etc., or SUN, MON, etc., if uppercase format characters are used)
dddd, DDDD	Display the full name of the day (e.g. Sunday, Monday, etc., or SUNDAY, MONDAY, etc., if uppercase format characters are used)
w	Display the day of the week as number (e.g. 1 for Sunday, ... , 7 for Saturday)
ww	Display the week of the year as number (e.g. 1, 2, ... , 52)
m	Display the month as number with no leading zero (e.g. 1, 2, ... , 12)
mm	Display the month as number with leading zero (e.g. 01, 02, ... , 12)
mmm, MMM	Display the name of the month in abbreviated form (e.g. Jan, Feb, etc., or JAN, FEB, etc., if uppercase format characters are used)
mmmm, MMMM	Display the full name of the month (e.g. January, February, etc., or JANUARY, FEBRUARY, etc., if uppercase format characters are used)
q	Display the quarter of the year (e.g. 1, 2, 3, 4)
y	Display the day of the year (1, 2, ... , 366)
yy	Display the year as 2-digit number (00, 01, ... , 99)
yyyy	Display the year as 4-digit number (100, 101, ... , 9999)
h	Display the hour as a number with no leading zero (0, 1, ... , 23)
hh	Display the hour as a number with leading zero (00, 01, ... , 23)
n	Display the minute as a number with no leading zero (0, 1, ... , 59)
nn	Display the minute as a number with leading zero (00, 01, ... , 59)
s	Display the second as a number with no leading zero (0, 1, ... , 59)
ss	Display the second as a number with leading zero (00, 01, ... , 59)
am, AM	Use the 12-hour clock and display am, pm (or AM, PM if uppercase format characters are used)

Table 2.6. *Date-time format specification characters*

For example, let us consider the following format specifications for the date-time November 8, 1957 at 3:45:33 PM:

```
"dd-MMM-yy"           08-NOV-57
"dd-mmm-yy"           08-Nov-57
"mmm d, yyyy"         Nov 8, 1957
"mmmm d, yyyy"        November 8, 1957
"hh:nn:ss"            15:45:33
"hh:nn AM"            03:45 PM
"dddd d, mmmm yyyy"   Friday 8, November 1957
"w, ww, y, q"         6, 45, 312, 4
```

Observe that this date-time was a Friday, in the 45th week, the 312th day of the last quarter of the year 1957.

For numeric expressions the *format_spec* must be a combination of the following format characters:

Numeric Format Spec	Description
0	Digit placeholder; displays a digit or zero
#	Digit placeholder; displays a digit or nothing
.	Decimal placeholder; determines how many digits to be displayed after the decimal point
%	Percentage placeholder; the expression is multiplied by 100 and the percent symbol is inserted at the location of the % character in the format specification
,	Thousand separator; indicates to format the number with thousand separators when used in front of the format specification. Otherwise when used at the end of the specification string it rounds the number in thousands
E-, E+	Scientific format; indicates to format the number in scientific notation
X, x	Hex digit place holder, converts an integer to its hexadecimal code with leading zeros

Table 2.7. *Numeric format specification characters*

Here are some examples of format specification string for the numeric (say `float` data type) value 5238983.7654387:

```
""                  5238983.765439
"00000000"          05238984
"$0.00"             $5238983.77
"0.00"              5238983.77
"0.000000"          5238983.765439
"#,###,##0"         5,238,984
"##0,,"             5
"#,"                5239
"0.00%"             523898376.54%
"0.00E+00"          5.24E+06
"00.00E-00"         52.39E05
"XXXXXX"            4FF0C7
"xxxxxxxx"          004ff0c7
```

Finally, for character string expressions the *format_spec* must be a combination of the following format characters:

String Format Spec	Description
@	Character place holder; displays a character or space, this placeholders are filled from right to left, to force left to right precede the string with the ! character
?	Character place holder; displays a character or the HTML character-code for space, this placeholders are filled from right to left, to force left to right precede the string with the ! character
&	Character place holder; displays a character or nothing, this placeholders are filled from right to left, to force left to right precede the string with the ! character
<	Force lower case characters
>	Force upper case characters
!	Force left to right fill of placeholders

Table 2.8. *String format specification characters*

For example, the following format specifications yield the corresponding effect on the right:

```
Format("Nemo", "@@@@@@@@@@@@@@")          "          Nemo"
Format("Nemo", "&&&&&&&&&&&")              "Nemo"
Format("Nemo", "!@@@@@@@@@@@@@@")          "Nemo          "
Format("Nemo", ">&&&&&&&&&&&&")            "NEMO"
```

Note that since in HTML standard the space characters are ignored (with the exception of one space), the *Format* function internally returns " " for each space placeholders provided we use the "?" format specification symbol instead of the "@" symbol. So for example, the `Format("Nemo", "??????????????")` returns the value:

```
"          Nemo"
```

which in HTML has the effect of forcing the space character padding.

Although, the SCRIPT engine implicitly converts a valid date-time values defined by character strings into DATETIME data type, nevertheless, it is sometimes necessary to explicitly convert a character string or numeric value into DATETIME data type value. In this case, the SCRIPT language has the *ToDate* conversion function, which has the following syntax:

```
DATETIME ToDate(expression, format_spec)
```

where *expression* is any valid expression of character string or numeric type, and *format_spec* is defined by a combination of format specification characters that depend on the data type of the expression. For character string expressions, the *format_spec* must be a combination of the following format specification characters:

Format Spec	Description
D	Expected the day as number with no leading zero (1, 2, ... , 31)
Dd	Expected the day as number with leading zero (01, 02, ... , 31)
M	Expected the month as number with no leading zero (1 through 12)
Mm	Expected the month as number with leading zero (01 through 12)
mmm, MMM	Expected the name of the month in abbreviated form (Jan, Feb, etc.)
mmmm, MMMM	Expected the full name of the month (January, February, etc.)
Yy	Expected the year as 2-digit number (00, 01, ... , 99)
Yyyy	Expected the year as 4-digit number (100, 101, ... , 9999)
H	Expected the hour as number with no leading zero (o, 1, ... , 23)
Hh	Expected the hour as number with leading zero (o0, 01, ... , 23)
N	Expected the minute as number with no leading zero (0, 1, ... , 59)
Nn	Expected the minute as number with leading zero (00, 01, ... , 59)
S	Expected the second as number with no leading zero (0, 1, ... , 59)
Ss	Expected the second as number with no leading zero (00, 01, ... , 59)
am, AM	Expected the AM or PM in the 12-hour clock

Table 2.9. *String format specification for ToDate function*

For example, for the following input date-time values the expected format specification string is given on the right:

```
"07-JAN-03"                 "dd-MMM-yy"
"07-Jan-2003"               "dd-mmm-yyyy"
"January 7, 2003"           "mmmm d, yyyy"
"January 7, 2003 13:33:05"  "mmmm d, yyyy hh:nn:ss"
"01/07/03 01:33:05 PM"      "mm/dd/yy hh:nn:ss AM"
```

Obviously, given the capability of the SCRIPT engine for the implicit conversion of date-time values from string expressions to DATETIME data type, the *ToDate* function with the above outlined format specification is not very useful. In the case of numeric expressions however, the *ToDate* function is more useful. In this later case the format specification is given by the following enumeration:

Format Spec	Description
YY or year	Will interpret the numeric value as the number of years starting from the initial date 01/01/1900.
QQ or quarter	Will interpret the numeric value as the number of quarters starting from the initial date 01/01/1900.

MM or month	Will interpret the numeric value as the number of months starting from the initial date 01/01/1900.
WK or week	Will interpret the numeric value as the number of weeks starting from the initial date 01/01/1900.
DD or day	Will interpret the numeric value as the number of days starting from the initial date 01/01/1900.
HH or hour	Will interpret the numeric value as the number of hours starting from the initial date 01/01/1900.
MI or minute	Will interpret the numeric value as the number of minutes starting from the initial date 01/01/1900.
SS or second	Will interpret the numeric value as the number of seconds starting from the initial date 01/01/1900.
MS or millisecond	Will interpret the numeric value as the number of milliseconds starting from the initial date 01/01/1900.

Table 2.10. *Format specification for ToDate function in the case of numeric expressions*

For the numeric value 3141 the following example illustrates the usage:

```
year          5040/12/30 00:00:00.000
month         2161/09/30 00:00:00.000
YY            5040/12/30 00:00:00.000
QQ            2685/03/30 00:00:00.000
MM            2161/09/30 00:00:00.000
WK            1960/03/12 00:00:00.000
DD            1908/08/06 00:00:00.000
HH            1900/05/09 21:00:00.000
MI            1900/01/01 04:21:00.000
SS            1899/12/30 00:52:21.000
MS            1899/12/30 00:00:03.141
```

Finally, we should remark that the format specification argument of the *ToDate* function is optional and may be skipped. In which case, the default interpretation will be assumed. This default format interpretation is "mm/dd/yyyy" for character string expressions, and DD for numeric expressions. For example, ToDate(3141) will be interpreted as ToDate(3141, DD).

The SCRIPT language has a conversion function similar to SQL Server, namely the *Cast* function. This function has the following syntax:

```
Cast(expression AS data_type)
```

where *expression* may be an expression of any data type, and the returned value will have data type specified by the *data_type*. For example,

```
DECLARE @nVar INT;

SET @nVar = Cast("5637" AS INT);
SET @nVar = @nVar + 32;   // @nVar now has the integer value of 5669
```

We should emphasize that the SCRIPT engine has complete set of internal conversion mechanisms, which in most cases succeeds in implicit conversion of data values between compatible data types. For example, the above result could have been achieved with the following alternative method:

```
DECLARE @nVar INT;

SET @nVar = "5637";
SET @nVar = @nVar + 32;   // @nVar now has the integer value of 5669
```

In some ambiguous circumstances though, the *Cast* function may come very handy.

Functions and Macros

The SCRIPT language has a rich collection of functions, which may be organized into the following categories:

- String functions,
- Date/time functions,
- Conversion functions,
- Mathematical functions,
- Special functions.

In this section, we will list these functions together with their respective definitions and usage. We start this listing with string functions:

`ASCII(sExpression)` — this function returns an integer representing the character code corresponding to the first letter in the character string `sExpression`.
Usage: `ASCII("A")` which equals to 65.

`CHR(nCharCode)` — this function returns the character corresponding to the character code specified by the argument `nCharCode`. Note that the return is always an ASCII character, which may not be supported in the HTML standards. To force the returned to be a string representing the HTML character code equivalent to the desired ASCII character, you may use the *H()* function.
Usage: `CHR(65)` which equals to "A".

`H(sExpression)` — this function translates the special ASCII characters in the string into HTML character codes.
Usage: `H(" ")` which equals to " ".

`ESCAPEURL(sString)` — this function encodes all URL unsave characters to the corresponding escape sequences.
Usage: `ESCAPEURL("a&b%#!'")` which equals to `"a%26b%25#!'"`.

`UNESCAPEURL(sString)` — this function decodes all URL encoded characters back to normal form.
Usage: `ESCAPEURL("a%26b%25#!'")` which equals to `"a&b%#!'"`.

`FORMAT(Expression, sFormatSpec)` — this function was already defined in the previous sections.

`LEFT(sString, nCount)` — this function returns a substring of `sString` of length `nCount`, starting from the left side of the string.
Usage: `LEFT("PM57U8938", 4)` which equals to "PM57"

`LEN(sString)` — this function returns the length of the string `sString`.
Usage: `LEN("PM57U8938")` which equals to 9.

`LOWER(sString)` — this function returns the value of `sString` in lower case characters.
Usage: `LOWER("CaLifoRnia")` which equals to "california".

`LTRIM(sString)` — this function returns the value of `sString` with left side trimmed off space characters.
Usage: `LTRIM("  BUSSINESS")` which equals to "BUSSINESS".

`REPEAT(sString, nCount)` — this function returns a string comprising of the string `sString` repeated `nCount` times.
Usage: `REPEAT("Sorry, ", 3)` equaling to "Sorry, Sorry, Sorry, ".

`REVERSE(sString)` — this function returns a string formed by reversing the characters of the string `sString`.
Usage: `REVERSE("Dracula")` which equals to "alucarD".

`RIGHT(sString, nCount)` — this function returns a substring of `sString` obtain by the `nCount` last characters.
Usage: `RIGHT("PM57U8938", 5)` which equals to "U8938"

`RTRIM(sString)` this function returns the value of `sString` with right side trimmed off space characters.
Usage: `RTRIM("BUSSINESS  ")` which equals to "`BUSSINESS`".

`SPACE(nCount)` this function returns a string comprising of "` `" repeated `nCount` times. Note that "` `" is the space character coding in HTML.
Usage: `SPACE(3)` which is equal to "`   `".

`TRIM(sString)` this function returns the value of `sString` with left and right sides trimmed off space characters.
Usage: `TRIM("  BUSSINESS  ")` which equals to "`BUSSINESS`".

`SUBSTR(sString, nPos, nLen)` this function returns the substring of `sString` starting at position `nPos` with length `nLen`.
Usage: `SUBSTR("PM57U8938", 3, 5)` which is equal to "`57U89`".

`FIND(sString, sFind, nStart)` this function returns the position of the first occurrence of the `sFind` string in the `sString` string, starting from position `nStart`.
Usage: `FIND("AbSced", "Sc")` which is equal to 2,
`FIND("abcfabed", "ab", 1)` which is equal to 4.

`REPLACE(sString, sFind, sReplace)` this function returns the string `sString` with all occurrences of the `sFind` string replaced with `sReplace` string.
Usage: `REPLACE("xyen xyich xyat","xy","wh")` which is equal to "`when which what`".

`UPPER(sString)` this function returns the value of `sString` in upper case characters.
Usage: `UPPER("CaLifoRnia")` which equals to "`CALIFORNIA`".

We outline next the date/time functions, but first we will need the following date-part enumeration table:

Date-Part	Abbreviation	Enumeration	Unit Type
Year	YY	1	Year
Quarter	QQ	2	Quarter
Month	MM	3	Month
Week	WK	4	Week
Dayofyear	DY	5	Day of year
Day	DD	6	Day
Weekday	DW	7	Week day
Hour	HH	8	Hour
Minute	MI	9	Minute
Second	SS	10	Second
Millisecond	MS	11	Millisecond

Table 2.11. *Date-part enumeration*

`DATEADD(DatePart, nCount, dtDate)` this function returns a date-time value based on adding an interval to the specified date-time `dtDate`. The `DatePart` argument determines what type of date-time units must be added, while `nCount` defines the interval. Possible values for `DatePart` are defined by the items `1` through `11` of the Table 2.11 except `5` and `7`, and may be specified in *Date-Part*, *Abbreviation* or *Enumeration* methods.
Usage: Given that `@dtV` is a variable of type `DATETIME`,
`DATEADD(DD, 3, @dtV)` to add 3 days to date-time `@dtV`,
`DATEADD(MM, 3, @dtV)` to add 3 months to date-time `@dtV`,
`DATEADD(HH, 3, @dtV)` to add 3 hours to date-time `@dtV`,
`DATEADD(week, 3, @dtV)` to add 3 weeks to date-time `@dtV`,
`DATEADD(2, 3, @dtV)` to add 3 quarters to date-time `@dtV`.

`DATEDIFF(DatePart, dtDate1, dtDate2)` this function returns the `DatePart` difference between the two date-time values as a decimal number. Again, possible values for

	DatePart are defined by the items 1 through 11 of the Table 2.11 except 5 and 7, and may be specified in *Date-Part*, *Abbreviation* or *Enumeration* methods. *Usage*: DATEDIFF(DD, @dtV1, @dtV2) to get the day difference between date-time @dtV1 and @dtV2.
DATENAME(*DatePart, dtDate*)	this function returns the *DatePart* name of a date-time value. Possible values for *DatePart* are defined by the items 1 through 11 of the Table 2.11, and may be specified in *Date-Part*, *Abbreviation* or *Enumeration* methods. *Usage*: If @dtV = "2003-03-21 20:56:42.550", then DATENAME(YY, @dtV) equals 2003, DATENAME(QQ, @dtV) equals 1, DATENAME(MM, @dtV) equals March, DATENAME(WK, @dtV) equals 12, DATENAME(DD, @dtV) equals 21, DATENAME(HH, @dtV) equals 20, DATENAME(MI, @dtV) equals 56, DATENAME(SS, @dtV) equals 42, DATENAME(MS, @dtV) equals 550, DATENAME(DW, @dtV) equals Friday, DATENAME(DY, @dtV) equals 80.
DATEPART(*DatePart, dtDate*)	this function returns the *DatePart* date-part of a date-time value. Possible values for *DatePart* are defined by the items 1 through 11 of the Table 2.11, and may be specified in *Date-Part*, *Abbreviation* or *Enumeration* methods. *Usage*: If @dtV = "2003-03-21 20:56:42.550", then DATEPART(YY, @dtV) equals 2003, DATEPART(QQ, @dtV) equals 1, DATEPART(MM, @dtV) equals 3, DATEPART(WK, @dtV) equals 12, DATEPART(DD, @dtV) equals 21, DATEPART(HH, @dtV) equals 20, DATEPART(MI, @dtV) equals 56, DATEPART(SS, @dtV) equals 42, DATEPART(MS, @dtV) equals 550, DATEPART(DW, @dtV) equals 6, DATEPART(DY, @dtV) equals 80.
GETDATE()	this function returns the current date-time of the system.
GETDAY(*dtDate*)	this function returns an integer representing the day of a date-time. *Usage*: If @dtV = "2003-03-21 20:56:42.550", then GETDAY(@dtV) equals 21.
GETMONTH(*dtDate*)	this function returns an integer representing the month of a date-time. *Usage*: If @dtV = "2003-03-21 20:56:42.550", then GETMONTH(@dtV) equals 3.
GETYEAR(*dtDate*)	this function returns an integer representing the year of a date-time. *Usage*: If @dtV = "2003-03-21 20:56:42.550", then GETYEAR(@dtV) equals 2003.
TODATE(*Expression, FormatSpec*)	this function converts an expression to date-time (it was described in **Format and Conversion Function** section of the current

chapter).

We now list the basic mathematical functions of the SCRIPT language:

`ABS(Value)` this function returns the absolute value of a numeric value.
Usage: `ABS(-3.22)` equals `3.22`.

`ACOS(Radians)` this function returns the arc-cosine of a numeric value given in radians between -1 and 1. For values outside this domain the function will return `NULL`.
Usage: `ACOS(0.9999)` equals `1.41422534775121E-02`.

`ASIN(Radians)` this function returns the arc-sine of a numeric value given in radians between -1 and 1. For values outside this domain the function will return `NULL`.
Usage: `ASIN(0.9999)` equals `1.55665407331738`.

`ATAN(Radians)` this function returns the arc-tangent of a numeric value in radians.
Usage: `ATAN(-45.01)` equals `-1.54858269620627`.

`CEILING(Value)` this function returns the smallest integer greater or equal to the numeric value.
Usage: `CEILING(10.58)` equals `11`.

`COS(Radians)` this function returns the cosine of a numeric value given in radians.
Usage: `COS(3.14)` equals `-0.99999873172754`.

`COT(Radians)` this function returns the cotangent of a numeric value given in radians.
Usage: `COT(3.14)` equals `-627.882397586913`.

`DEGREES(Radians)` this function converts the value of angle from radians to degrees.
Usage: `DEGREES(3.14)` equals `179.908899633625`.

`EXP(Value)` this function returns the exponent of a numeric value.
Usage: `EXP(34)` equals `583461742527455`.

`FLOOR(Value)` this function returns the greatest integer less than or equal to a numeric value.
Usage: `FLOOR(10.58)` equals `10`.

`LOG(Value)` this function returns the natural logarithm of a numeric value.
Usage: `LOG(3.22)` equals `1.16938135955632`.

`LOG10(Value)` this function returns the base-10 logarithm of a numeric value.
Usage: `LOG10(3.22)` equals `0.507855871695831`.

`PI()` this function returns the constant value `3.14159265358979`.

`POWER(Base, Exponent)` this function returns the power of the value of the *`Base`* to the *`Exponent`*.
Usage: `POWER(2, 3)` equals `8`.

`RADIANS(Degrees)` this function converts the value of the angle from degrees to radians.
Usage: `RADIANS(30)` equals `0.523598333333333`.

`RAND(MaxValue)` this function returns a random integer value between `0` and *`MaxValue`*.
Usage: `RAND(30)` equals a random value between `0` and `30`.

`ROUND(Value, Precision [,Option ])` this function rounds a numeric value to the prescribed *`Precision`* decimal places when *`Option`* is not specified equals or when it is specified as 0. When *`Option`* is 1, the function simply truncates to the decimal places specified by *`Precision`*.
Usage: `ROUND(3.45675, 4)` equals `3.4568`,

```
ROUND(3.45675, 4, 0) equals 3.4568,
ROUND(3.45675, 4, 1) equals 3.4567.
```

`SIGN(Value)` this function returns an integer value indicating the sign of an expression.
Usage: `SIGN(-920)` equals `-1`.

`SIN(Radians)` this function returns the sine of a numeric value given in radians.
Usage: `SIN(PI()/2)` equals `0.99999999999912`.

`SQRT(Value)` this function returns the square root of a numeric value.
Usage: `SQRT(2)` equals `1.4142135623731`.

`TAN(Radians)` this function returns the tangent of a numeric value given in radians.
Usage: `TAN(PI()/2)` equals `753695.99514081`.

Finally, we consider the rest of the special functions of SCRIPT language:

`PAGENUM()` this function returns the currently generating page number.
Usage: `PAGENUM()+ 2` equals 2 plus the current page number during the report generation.

`PAGECOUNT()` this function returns the total number of pages of the generated report output. Note that this special function must be used as a term by itself, it cannot be used as an operand term or an argument to another function.
Usage: `Page <FLD>PAGENUM()</FLD> of <FLD>PAGECOUNT()</FLD>`

`EVALEXPR(String)` this function returns the value of expression given as a string. The string argument is evaluated as expression at run-time.
Usage: In this example an expression is given as string and then evaluated by calling this
`EVALEXPR` function:

```
DECLARE @sExpression VARCHAR(100);
DECLARE @fResult FLOAT;

SET @sExpression = "SIN(PI()/2)*COS(PI()/2)";
SET @fResult = EVALEXPR(@sExpression); // evaluated here
```

Note that instead of the variable `@sExpression`, we could have a parameter option variable, which can pass various expressions from the host application to be evaluated at run-time by the SCRIPT engine.

`VALUEAT(Variable, Event)` this function returns the value of a pending variable, which is evaluated when the specified event occurs. Possible values for the second argument are as follows:

```
ELS_OnBeginPage,
ELS_OnEndPage,
ELS_OnBeginReport,
ELS_OnEndReport.
```

This function is very helpful in the implementation of running totals and summaries of complex grouping. We will describe more details about this function in the context of pending variables, in later sections of the current chapter.
Usage: `VALUEAT(@vVar, ELS_OnEndPage)` will evaluate the variable `@vVar` at the end of each page during the report generation.

`SENDMESSAGE(Value)` this function will raise an integer valued message in the host application through event handler API functions. It could be used as a progress indicator in the host application.
Usage: `SENDMESSAGE(0)` which will send a message with `0` identifier number.

`ISSET(Object)` this function returns true if the *`Object`* argument is a valid non-null object, otherwise it retruns false. The argument *`Object`* is any variable of object type category in SCRIPT.

Usage:
```
DECLARE @conn CONNECTION;
DECLARE @ds DATASOURCE;
...
IF ISSET(@conn) THEN
        @ds.Connect(@conn);
END IF
```

`COALESCE(Exp, AltExp)` this function returns *Exp* if this expression is not null, otherwise it returns the *AltExp* expression.

Usage:
```
SET @oN = @oNode.selectSingleNode("price");
SET @sPrice = COALESCE(@oN.text, "0.00");
```

Although the functions of the SCRIPT language are very much similar to the MS-SQL Server procedural language, and therefore are very intuitive to use, nevertheless, the user may utilize the **Expression Builder** window and **Format/ Conversion Wizard**, to define and insert functions into the SCRIPT *Editor*. The **Expression Builder** window essentially comprises of the following four panes (see Figure 2.22 for more details):

Fields/Variables which displays all the data fields and variables that are defined in the report script, so that the user may drag-drop them into the *Editor* pane.

Functions which displays all the SCRIPT functions categorized into mathematical, string, date-time, conversion and special functions. The user may drag-drop any of these functions into the *Editor* pane.

Editor which is a fully developed SCRIPT editor, on its own, furnished with *IntelliSense* support and basic edit operations.

Usage this pane is an online help window outlining the usage of the function selected in the **Functions** pane. It may be expanded or collapsed via the arrow button at the bottom-left corner of the pane.

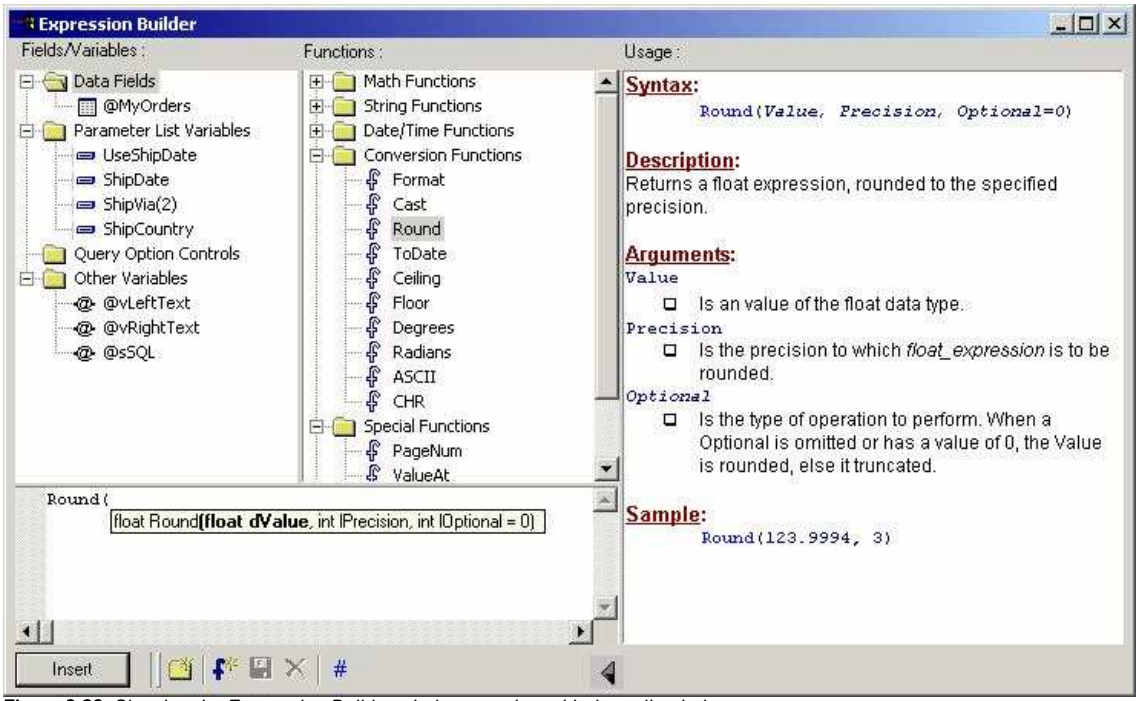


Figure 2.22. Showing the Expression Builder window together with the online help pane

The **Format/Conversion Wizard** dialog may be called from the **Expression Builder** window via the toolbar button with the #-sign icon. In this window the user may select a format specification by selecting a predefined format from the **Format** combo-box, or manually typing in the format specification character combination and click the *ENTER* key to check the result in the **Result Sample** display-field (see Figure 2.23 for more details).

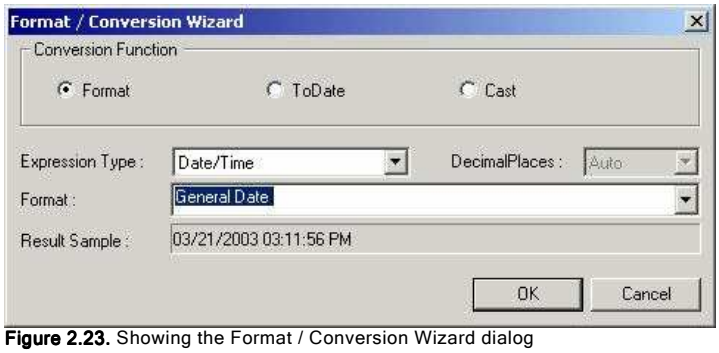


Figure 2.23. Showing the Format / Conversion Wizard dialog

The SCRIPT language has capabilities to define macros via pre-compile directives. Essentially, by pre-compile directive we mean that prior the report compilation a substitution is performed to expand all definitions into the primitives of the SCRIPT language and then perform compilation of the report. The pre-compile directives of the SCRIPT language are as follows: `#include`, `#define`, `#undef`, `#ifdef`, `#ifndef`, `#else` and `#endif`. We describe the usage of these directives next:

`#include` this is used to include the macros of an external *ELS*-file into the current script. The filename and path information following this keyword must be on the same line.
Usage: The following example will include the definitions of file ***MyMacros.txt*** in the current script

```
#include "THIS_FILE\Include\MyMacros.txt"
```

`#define` this is used to define a macro. The macro name must follow the keyword on the same line, but the definition may extend over multiple lines with each line ending with the "`\`" character.
Usage: The following example illustrates the usage:

```
#define MyFunction(X, Y, Z) X*X - 2*X*Y + Z
```

`#undef` this is used to undefine a macro name.

`#ifdef` this is used to check if a particular macro name is defined.

`#ifndef` this is used to check if a particular macro name is not defined.

`#else` this is used to check the else-condition of a conditional.

`#endif` this is used to end a conditional.

Note that pre-compile directives are intended to be used for short definitions or macros, it is not intended to be used as report template reuse or duplication purposes. For report template reuse, please consult the *Standard Report Template meta-Language* (*SRTmL*) or the *SRT Wizards* topics in Chapter 1 or technical articles web site.

Parameter Options

Most report engines have a fixed set of API functions, with which the developer may control various information presented in the report. The fact that such API functions are predefined and fixed by design, sometimes constrains the flexibility or amount of information in the report that may be controlled by the host application. In contrast, the SCRIPT engine has a fixed set of API functions to handle the report initialization and generation, but in regards to controlling the information in the reports, it is entirely user-definable. In fact, almost any element in a particular report script may be controlled via report parameter options defined inside the report script itself. Report parameter options are defined in a report script via the `PARAM_OPTIONS` construct, which can be declared only in the *ELS_QPARAMS* report section using the following syntax:

```
PARAM_OPTIONS
[
    variable_1 datatype_1,
    variable_2 datatype_2,
    ...
    variable_N datatype_N
];
```

where *`variable_i`* are the names of variables of data types *`datatype_i`*, respectively for $i = 1, 2, \ldots, N$. Note that

unlike regular variable names the report parameter options variable names do not start with the symbol "@". This convention was adopted to introduce some distinction between the names of regular variables and those of the parameter options. This syntax does not preclude the usage of arrays for the variable names, so that arrays of parameter option variables may be defined just like the regular variables.

In addition to the declaration section, a report parameter variable has the following optional properties:

`Nullable`	boolean type with default value *FALSE*, this indicates whether the parameter can assume *NULL* value.
`AllowBlank`	boolean type with default value *TRUE*, this applies to parameters of character data types only, and indicates whether the parameter is allowed to have empty string as a value.
`Prompt`	string type with default value equal to the name of the parameter itself, this will serve as a label text for the parameter when called from the host application.
`MultiValue`	boolean type with default value *FALSE*, this indicates whether multiple values can be assigned to the parameter from the host application. Note that if this property is set to *TRUE*, then you must declare the parameter variable as an array with the maximum possible number of values as the dimension of the array.
`DefaultValues`	variant type, this is used to define the default values for the parameter in the host application. It can either be a sequence of comma separated values between curly brackets, or a data source reference. We will describe more details about this property shortly.
`ValidValues`	variant type with default value *NULL*, this will define the content of a list in the host application from which valid values may be selected. Each item consists of a pair of label and data values. Just like the `DefaultValues` this property may also have a data source reference for a value. We will see more details later on in this section.
`SourceType`	integer type with default value 0, indicates whether the `ValidValues` property is a list or a data source reference. Possible values are 0 for list and 1 for data source.
`Value`	variant type, this is used to set or get the parameter's value. This property is the default property of the parameter.

Before describing more details about `DefaultValues` and `ValidValues` properties, we will first illustrate the usage of report parameter options in some generic sample code in SCRIPT. In particular, the following code lines should illustrate the usage of multi-valued parameters with explicitly listed valid values and default values:

```
<ELS_QPARAMS>
    PARAM_OPTIONS
    [
        var0        bit,
        var1        varchar(50),
        var2        int,
        var3(2)     varchar(30)
    ];

    Var0.Prompt = "Have Insurance: ";
    Var0.DefaultValues = {1};

    var1.Prompt = "Last Name: ";
    var1.AllowBlank = FALSE;
    var1.DefaultValues = {"Smith"};

    var2.Prompt = "Family Size: ";
    var2.Nullable = TRUE;
    var2.ValidValues = {("No kids", 2), ("One kid", 3),
                        ("Two kids", 4), ("Three kids", 5),
                        ("Four kids", 6), ("Five kids", 7),
                        ("Half a dozen kids", 8)};
    var2.DefaultValues = {3};

    var3.Prompt = "Select Options: ";
```

```
    var3.AllowBlank = FALSE;
    var3.MultiValue = TRUE;
    var3.ValidValues = {("Hearth Checkup","HC"),
                        ("Diabetics Analysis","DA"),
                        ("General Checkup","GC")};
    var3.DefaultValues = {"DA", "HC"};
</ELS_QPARAMS>
```

Observe that when we compile and run the report script that contains such listing of parameter options, the following dialog prompts in the *Report Designer* (see Figure 2.24):

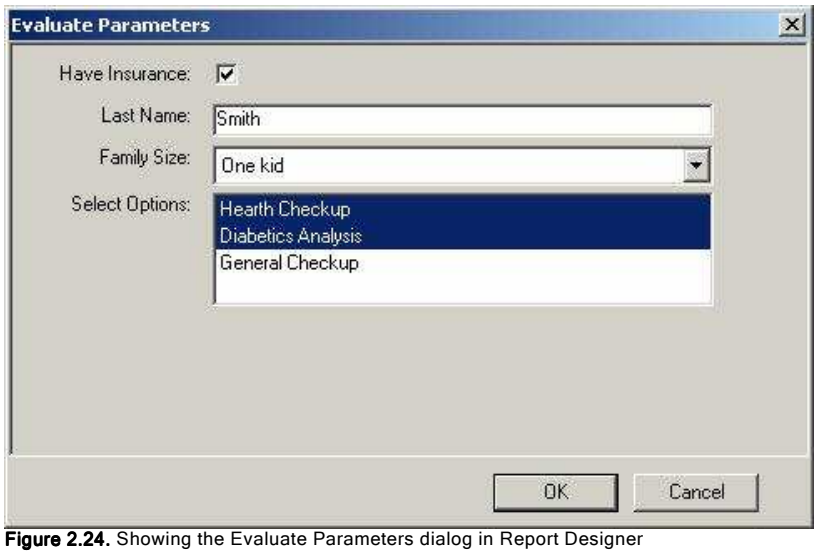


Figure 2.24. Showing the Evaluate Parameters dialog in Report Designer

Note that, in particular the `Prompt` property values appear as labels for the parameter options in the **Evaluate Parameters** dialog. The `ValidValues` property values fill the combo-box and list-box respectively for the `var2` and `var3` variables. In fact, if a parameter can assume multi-values then it should be interpreted as a list-box, otherwise it should be interpreted as a combo-box. Finally, observe that the valid values are defined via pairs, with the first coordinate being the label to be displayed in the control of the host application, while the second coordinate holds the actual value of the parameter to be passed from the host to the report engine.
In general, the value format of the `DefaultValues` property is as follows:

```
    variable.DefaultValues = {val_1, val_2, ... , val_N};
```

where `val_j` are values of data in the parameter's data type (in particular, for single-valued parameters $N = 1$).
The value format of the `ValidValues` property is as follows:

```
    variable.DefaultValues = { (label_1, val_1), (label_2, val_2), ... , (label_M, val_M) };
```

where `label_j` are label texts and `val_j` the actual possible valid values.

In the previous code sample, we have assumed that default values and valid values are to be passed via explicit lists. In fact, since the `SourceType` property was not specified the default value of 0 was assumed, which instructed the report engine to construct the default values and valid values collections via the explicit list of items.
As we have mentioned that default values and valid values of report parameters may be alternatively specified via data source references. In this case, the `SourceType` property must be set to 1, and a `DATASOURCE` object must be defined prior to the definition of the parameter. The following code sample illustrates this usage:

```
<ELS_QPARAMS>
    PARAM_OPTIONS
    [
        var4 varchar(150)
    ];

    // define a data source
```

```
    DECLARE @dsEmployees DATASOURCE;
    SET @dsEmployees = "SELECT lastname + ', ' + firstname EmployeeName, EmployeeID " +
                       "FROM employees";

    var4.SourceType = 1;  // use data source reference
    var4.Prompt = "Employee: ";
    var4.ValidValues = (@dsEmployees, "EmployeeName", "EmployeeID");
    var4.DefaultValues = (@dsEmployees, "EmployeeName", "EmployeeID");
</ELS_QPARAMS>
```

To define parameter options for a report, the user may edit directly in the SCRIPT *Editor* (i.e. **Source** view) with the help of *IntelliSense*, or alternatively use the **Parameter List** sliding window to visually define the parameters along with respective properties (or both methods). To slide-on the **Parameter List** window, simply click on the **Parameter List Window** toolbar button in the **Project Explorer** pane. Figure 2.25 depicts the **Parameter List** window for the sample code that used explicitly listed valid values:

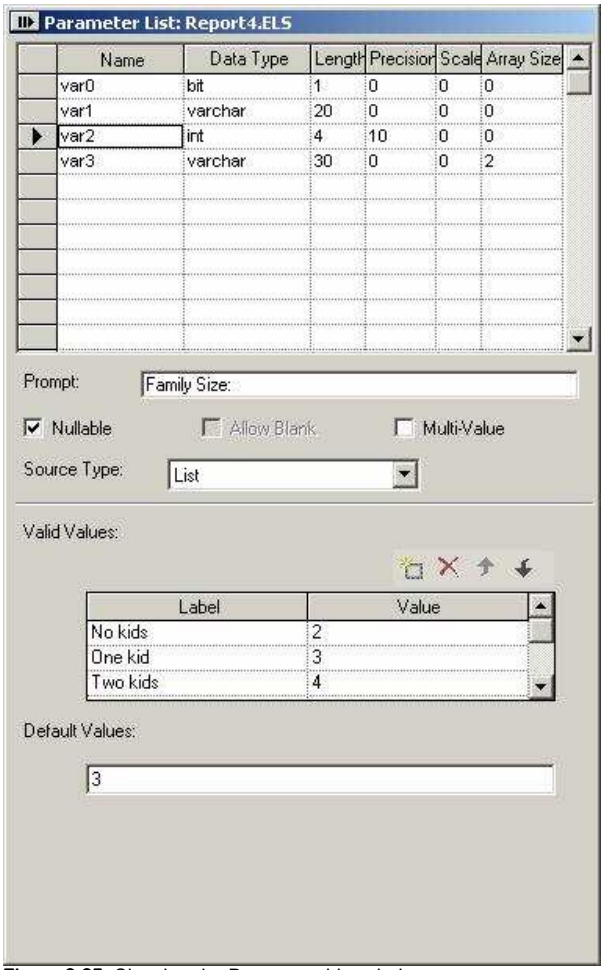


Figure 2.25. Showing the Parameter List window

Essentially, this sliding window consists of an upper grid and extended controls lower section. The upper grid consists of the following columns:

Name to enter the parameter option variable name,

Data Type to select the data type for the parameter option variable,

Length to enter or display the length of the data type,

Precision to enter or display the precision of the data type (if applicable),

Scale to enter or display the scale of the data type (if applicable),

Array Size to enter the array size if we are defining an array of the variable.

Note that the **Precision** and **Scale** are applicable only for numeric variables, while the **Length** is editable only when defining character strings.

The appearance of the lower section of the **Parameter List** window may vary depending on the data type of the selected variable in the upper grid. Note that the controls of this lower section are used to define the extended properties of the parameter variable that is selected in the upper grid.

In particular, if the parameter variable that is selected in the upper grid of the **Parameter List** window uses a list source type, then the controls in the lower section of the window will look like those in Figure 2.25. In general, these controls are as follows:

Prompt edit-box to enter the value for the `Prompt` property of the selected parameter.

Nullable check-box to define the value for the `Nullable` property of the selected parameter.

Allow Blank check-box to define the value for the `AllowBlank` property of the selected parameter.

Multi-Value check-box to define the value for the `MultiValue` property of the selected parameter.

Source Type combo-box to define the value for the `SourceType` property of the selected parameter.

Valid Values this control depends on the value selected in the **Source Type** and may be a grid with **Add-Delete** toolbar, or a series of combo-boxes (we will describe more details shortly).

Default Values this control depends on the values selected in **Source Type** and **Multi-Value** and may be an edit-box, a grid with **Add-Delete** toolbar, or a series of combo-boxes (we will describe more details shortly).

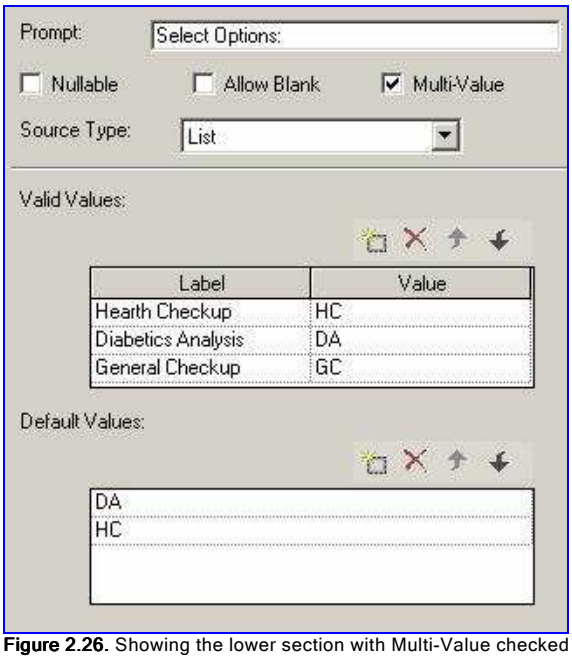


Figure 2.26. Showing the lower section with Multi-Value checked

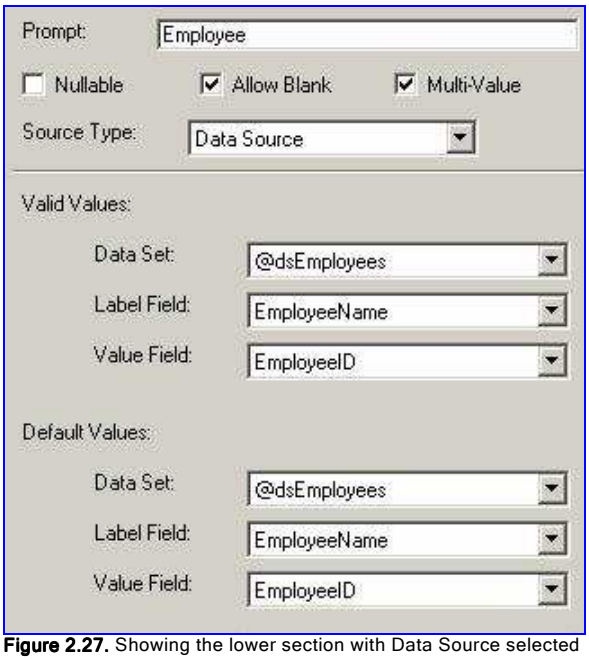


Figure 2.27. Showing the lower section with Data Source selected

Now, if the user selects List in the **Source Type** combo-box and makes the **Multi-Value** check-box unchecked, then the **Valid Values** control will become a grid with two columns **Label** and **Value**. On the upper-left corner of this list you will see a small toolbar with the buttons **Add**, **Delete**, **Move Up** and **Move Down**. These buttons are used to add a new row to the grid, to delete an existing row from the grid, or to move a row up or down one step. The **Default Values** control on the other hand, will be a simple edit-box, in which the user may enter the value of the `DefaultValues` property for the selected parameter.
Now, if the user makes the **Multi-Value** check-box checked, then the **Default Values** control will become a single columned grid with the **Add**, **Delete**, **Move Up** and **Move Down** toolbar. This later case is depicted in Figure 2.26.

Finally, in the case when the **Source Type** control is set to Data Source, each of the **Valid Values** and **Default Values** controls will become a combination of the following three combo-box controls:

Data Set to define the data source reference,

Label Field to define the display field for the parameter control in the host,

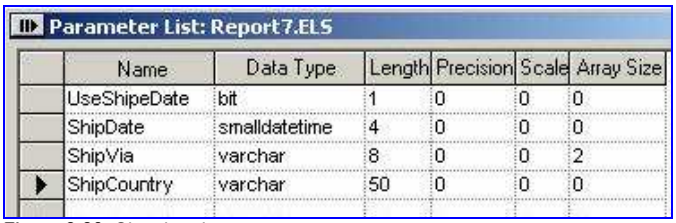
Value Field to define the actual value field for the parameter.

In using the **Parameter List** window, observe that whenever this window becomes out of focus, it will automatically slide-off, and the pending changes will update the *ELS_QPARAMS* section of the report.

Getting back to the **Source** view of the report script, note that, although the parameter options can only be declared inside the *ELS_QPARAMS* section of a report, these parameters can be referenced from anywhere in the report script just like any pure *ELS*-element. In this way, the parameter options behave like a user-definable interface between elements inside a report script and the host application that needs to evaluate these parameter options. Moreover, as we will see in Chapter 3, that the entire structure of the parameter options of a report may be exposed to the host application via the *GetParams()* API function.

To illustrate the full usage of parameter options we make a copy of the Report2.ELS in the current report project,

and then rename it as Report7.ELS. Now open this Report7.ELS file into the SCRIPT *Editor*, and perform the following modifications.



Name	Data Type	Length	Precision	Scale	Array Size
UseShipeDate	bit	1	0	0	0
ShipDate	smalldatetime	4	0	0	0
ShipVia	varchar	8	0	0	2
ShipCountry	varchar	50	0	0	0

Figure 2.28. Showing the

First, slide-on the **Parameter List** window by selecting the **Parameter List** menu item under the **Tools** menu. For our sample, we need to define four parameter option variables: *UseShipDate*, *ShipDate*, *ShipVia* (array of size 2) and *ShipCountry*. After entering the information into the upper grid of the **Parameter List** window (see Figure 2.28), click the slide-off arrow button on the top-right side of the window, to slide-off the **Parameter List** window. This action will insert the following parameter options in the *ELS_QPARAMS* section of the report script:

```
PARAM_OPTIONS
[
        UseShipDate     bit,
        ShipDate        smalldatetime,
        ShipVia(2)      varchar(8),
        ShipCountry     varchar(50)
];
ShipVia.DefaultValues = {""};
ShipCountry.DefaultValues = {""};
```

The plan here is that we intend to pass parameters from the host application into this report, informing it whether we want to use *ShipDate* as parameter, and if so what this date's value must be, or else what *ShipVia* and *ShipCountry* values we want to pass to this report. So that, based on these parameter values the *Orders* report will be generated.

To continue with the evaluation of the properties of the parameters switch back to the **Parameter List** window via the slide-on button. In this window, select the `UseShipDate` parameter and set the **Prompt** control's value to "`Use Ship Date: `", also enter 1 in the **Default Values** edit-box. Similarly select the `ShipDate` parameter and set the **Prompt** value to "`Shipped Date: `", and the **Default Values** to "`01-01-1997`". Now slide-off the **Parameter List** window to update the report script. This action will add the following additional lines:

```
UseShipDate.Prompt = "Use Ship Date: ";
UseShipDate.DefaultValues = {1};
ShipDate.Prompt = "Shipped Date: ";
ShipDate.DefaultValues = {"01-01-1997"};
```

Now, let us assume that the user requirements for the `ShipVia` and `ShipCountry` parameters must have selectable lists in the host application, so that the end-user will simply select values from a predefined list of items instead of direct keyboard entry. Moreover, for both of these parameters we want to use data source references. For example, add the following *Shippers* data source declaration at the end of the parameter properties in the *ELS_QPARAMS* section of the **Source** view:

```
DECLARE @Shippers DATASOURCE;
SET @Shippers = "SELECT * FROM Shippers";
```

Then click the slide-on button to slide on the **Parameter List** window. In this window, select the `ShipVia` parameter in the upper grid, then set the **Prompt** value to "`Ship Via: `", check the **Multi-Value** check-box, and set the **Source Type** to Data Source. Note that the **Valid Values** and **Default Values** controls change to **Data Set**, **Label Field** and **Value Field** combo-box combinations. For both properties respectively set the **Data Set** to @Shippers value, the **Label Field** to CompanyName and **Value Field** to ShipperID. Sliding off the **Parameter List** window will add the following lines in the *ELS_QPARAMS* section:

```
ShipVia.Prompt = "Ship Via: ";
ShipVia.MultiValue = TRUE;
ShipVia.SourceType = 1;
ShipVia.ValidValues = (@Shippers, "CompanyName", "ShipperID");
```

```
ShipVia.DefaultValues = (@Shippers, "CompanyName", "ShipperID");
```

Similarly, for `ShipCountry` parameter, add the following data source to the end of the *ELS_QPARAMS* section and slide on the **Parameter List** window:

```
DECLARE @ShipCountries DATASOURCE;
SET @ShipCountries = "SELECT DISTINCT ShipCountry FROM Orders";
```

In the upper grid of the **Parameter List** window select the `ShipCountry` parameter and set the **Prompt** control to "`Ship Country:` ", the **Source Type** to Data Source and for both the **Valid Values** and **Default Values** respectively set the **Data Set** to @ShipCountries, **Label Field** to ShipCountry, and **Value Field** to ShipCountry. Sliding off the **Parameter List** window will add the following lines at the end of the *ELS_QPARAMS* section:

```
ShipCountry.Prompt = "Ship Country: ";
ShipCountry.SourceType = 1;
ShipCountry.ValidValues = (@ShipCountries, "ShipCountry", "ShipCountry");
ShipCountry.DefaultValues = (@ShipCountries, "ShipCountry", "ShipCountry");
```

In the Report7.ELS, after further manual editing in the *ELS_QPARAMS* and ELS_RSETTINGS sections we obtain the following SCRIPT code:

```
<ELS_QPARAMS>
    // parameter options
    PARAM OPTIONS
    [
           UseShipDate bit,
           ShipDate smalldatetime,
           ShipVia(2) varchar(8),
           ShipCountry varchar(50)
    ];

    UseShipDate.Prompt = "Use Ship Date: ";
    UseShipDate.DefaultValues = {1};
    ShipDate.Prompt = "Shipped Date: ";
    ShipDate.DefaultValues = {"01-01-1997"};

    DECLARE @Shippers DATASOURCE;
    SET @Shippers = "SELECT * FROM Shippers";

    ShipVia.Prompt = "Ship Via: ";
    ShipVia.MultiValue = TRUE;
    ShipVia.SourceType = 1;
    ShipVia.ValidValues = (@Shippers, "CompanyName", "ShipperID");
    ShipVia.DefaultValues = (@Shippers, "CompanyName", "ShipperID");

    DECLARE @ShipCountries DATASOURCE;
    SET @ShipCountries = "SELECT DISTINCT ShipCountry FROM Orders";

    ShipCountry.Prompt = "Ship Country: ";
    ShipCountry.SourceType = 1;
    ShipCountry.ValidValues = (@ShipCountries, "ShipCountry", "ShipCountry");
    ShipCountry.DefaultValues = (@ShipCountries, "ShipCountry", "ShipCountry");
</ELS_QPARAMS>

<ELS_RSETTINGS>
    SET REPORT_TITLE      = "Report Title";
    SET PAGE_ORIENTATION  = ELS_PORTRAIT;
    SET PAGE_SIZE         = ELS_LETTER;
    SET PAGE_SOURCE       = ELS_UPPER;
    SET PAGE_MARGINS.LEFT        = 0.5;
    SET PAGE_MARGINS.RIGHT       = 0.5;
    SET PAGE_MARGINS.TOP         = 0.75;
    SET PAGE_MARGINS.BOTTOM      = 0.5;
    SET DEFAULTMODE              = ELS_FAST;
    SET SUPPRESS_PHEADER.FIRSTPAGE = TRUE;
    SET SUPPRESS_PFOOTER.FIRSTPAGE = TRUE;

    DECLARE @MyOrders DATASOURCE;
    DECLARE @sSQL VARCHAR(1000);
```

```
    // define the @sSQL string depending on the value of the parameter options
    SET @sSQL = "SELECT " +
                    "OrderID, " +
                    "OrderDate, " +
                    "ShippedDate, " +
                    "ShipName, " +
                    "ShipAddress " +
            "FROM " +
                    "Orders ";
    // if UseShipDate is true then use ShipDate value and ignore the other parameters,
    // otherwise use the values of the other parameters
    IF UseShipDate = 1 THEN
            SET @sSQL = @sSQL + "WHERE ShippedDate > '" + Format(ShipDate, "mm-dd-yyyy")+ "'";
    ELSE
            DECLARE @v varchar(100);
            DECLARE @i int;

            SET @v = "ShipVia = " + ShipVia(@i);
            SET @i = @i + 1;
            WHILE @i < 2   // we are assuming that the user can select at most 2 ShipVia-s
                    IF ShipVia(@i) <> "" THEN
                            SET @v = @v + " OR ShipVia = " + ShipVia(@i);
                    END IF
                    SET @i = @i + 1;
            END LOOP

            SET @sSQL = @sSQL + "WHERE (" + @v + ") AND ShipCountry LIKE '" + ShipCountry + "'";
    END IF
    // define the DATASOURCE variable by setting it to the value of @sSQL
    SET @MyOrders = @sSQL;
</ELS_RSETTINGS>
```

Observe that in the code above we have defined the *WHERE*-clause of the SQL command via the string `@sSQL` variable in a conditional manner depending on the values of the parameter options. In particular, for the construction of the SQL command, we have deviced a loop to consider all the values of the multi-valued `ShipVia` parameter. Note that, in this example we have assumed that the user will select a maximum of 2 items from the multivalued **ShipVia** list-box. To be able to select more than 2 values, you need to increase the array size, as well as adjust the condition of the `WHILE` loop.

Eventually, the values of the parameter options are intended to be passed via the host application, and in this way, the data source will be controlled from the host application. But in the absence of such a host application, the *Report Designer* will prompt the user with the **Evaluate Parameters** dialog whenever the compiled report containing parameter options is triggered to generate report output. In the case of the current report example, when after compilation we trigger the report generation, the **Evaluate Parameters** dialog prompts with the parameter options displayed as in Figure 2.29 below:

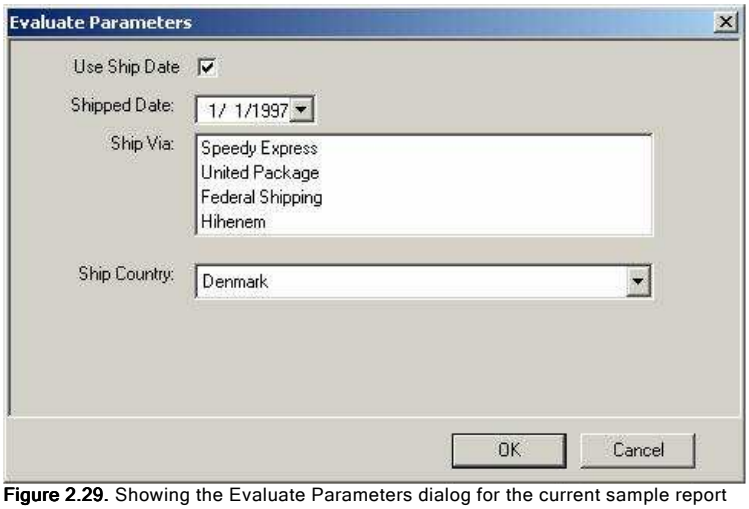


Figure 2.29. Showing the Evaluate Parameters dialog for the current sample report

The user may then enter desired values for the parameter options shown in this dialog to check or debug the report output. Therefore, as we can obviously see that the **Evaluate Parameters** dialog indeed acts like a handy

debugging tool to test the generation of a report that contains parameter options.

Query Options

NOTE: In version 3.0 of ELS-Script software, the GEV-engine became obsolete, therefore this section is currectly unsupported. Moreover, versions of ELS-Script after 3.0 may not support GEV-engine. This section is retained in this document for backward compatibility purposes only. If you are using version 3.0 or above, please skip this section and jump to section about "Pending Variables".

There are two versions of the SCRIPT engine. The **(G|E|V)**-version, which essentially comes with a built-in **Report Generator** and **Report Viewer**, and the **(E)**-version, which is a much lighter version containing only the core elements of the report engine. Query options are applicable only when using the **(G|E|V)**-version of the SCRIPT engine, and essentially instruct the **Report Generator** window how to construct a prescribed query form for the selected report, for example, the various types of controls that may be used, along with respective properties and events interrelating them in the query form.

Some of the major advantages of utilizing the **Report Generator** window are as follows:

1. Dispensing the need to develop special query forms in the host application, since such information may be included in the report file itself.
2. Adding new reports without recompiling the host application, since query form information, as well as, data source definitions may be imbedded inside the report files.
3. Simplified integration of the host application with the backend report engine.
4. Self-contained and uniform report development platform with a universal programming language, namely the SCRIPT language.

In fact, the SCRIPT language being very similar to MS-SQL Server procedural language unifies the database backend, the report scripting, as well as, the front host application's query forms into a single universal programming platform, while its scripting nature extends its flexibility, relentlessly integrating it into DHTML, ASP and ASP.NET.

Our discussion of the query options will proceed first with an exposition of the syntax of declaration and definition, and then the description of the **Query Form** designer, followed with a sample report illustrating the general usage of query options.

Query options must be declared only in the *ELS_QPARAMS* report section, and the general declaration syntax is as follows:

```
QUERY_OPTIONS
[
    variable_1 controltype_1,
    variable_2 controltype_2,
    ...
    variable_N controltype_N
];
```

where `variable_i` are valid variable names of control types `controltype_i`, respectively for $i = 1,2, \ldots, N$. The possible control types for query options, which cover almost all Windows standard GUI controls, are listed below:

`STATIC_CONTROL`	which defines a static control,
`TEXTBOX_CONTROL`	which defines a textbox control,
`CHECKBOX_CONTROL`	which defines a checkbox control,
`OPTION_CONTROL`	which defines a radio-button control,
`LISTBOX_CONTROL`	which defines a list-box control with an **All Items** checkbox,
`COMBOBOX_CONTROL`	which defines a combo-box control,
`SPIN_CONTROL`	which defines a spin control,
`DATE_CONTROL`	which defines a date control,
`DATERANGE_CONTROL`	which defines a date range control,
`NUMBERRANGE_CONTROL`	which defines a number range control,
`GRID_CONTROL`	which defines a grid control,
`GROUP`	which defines a group frame,

`SEPARATE BY` which defines a specified separator space,

A control variable of one of these control types has inherent properties and events depending on the type of the control. Some properties and events that are common to all but the last two control types are listed below:

Property or Event	Description
`VariableName`	This property retrieves the variable name of a control variable. It is read-only.
`Name`	This property's value becomes the caption name of the control which is displayed as a label text on the left or top side of the control. If this property is not specified the caption of the control will be the control's name by default.
`Operators`	This property is used to define a list of operators to be used in a combo-box with the control, by default this list is empty and no operator combo-box should appear preceding the control. Possible operators are: `=`, `<`, `>`, `<=`, `>=`, `<>`, `LIKE`, `NOT LIKE`, `BETWEEN`, `NOT BETWEEN`, `IN` and `NOT IN`.
`DefaultOperator`	This applies only if the operator list is non-empty, in which case defines the default operator that will be displayed in the operator combo-box.
`Height`	This property defines the height of the control in pixels, by default the height of any control will be the internally defined default height of the control.
`Width`	This property defines the width of the control in pixels, by default the width of any control will be the internally defined default width of the control.
`Alignment`	This property defines the alignment of the control, by default the control will be left aligned. Possible alignments are: `ELS_LEFT`, `ELS_CENTER` and `ELS_RIGHT` respectively for left, centered and right alignments.
`LabelStyle`	This property defines the label or caption text position, possible styles are `ELS_LEFT` for labels positioned on the left of the control, and `ELS_TOP` for labels positioned on the top of the control. By default labels are put on the left.
`Visible`	This property defines visibility of the control, by default the visibility is true.
`Enabled`	This property makes the control enabled or disabled, by default the control is enabled.
`OnChange`	This event is triggered when the value of the control is changed at run-time, in which case any SCRIPT code included in this event handler will be executed, controlling the behavior of the contents of the query form.
`OnGotFocus`	This event is triggered when the control gets in focus at run-time, in which case any SCRIPT code included in this event handler will be executed, controlling the behavior of the contents of the query form.
`OnLostFocus`	This event is triggered when the control loses focus at run-time, in which case any SCRIPT code included in this event handler will be executed, controlling the behavior of the contents of the query form.

Table 2.12. *Common properties and event handlers*

Setting or defining property values are allowed only inside the *ELS_QPARAMS* report section. In contrast, these values may be called or used inside *ELS* and *FLD* tags located anywhere in the entire report script. The event handlers `OnChange`, `OnGotFocus` and `OnLostFocus`, may be defined only inside *ELS-QPARAMS* section, and have the following syntax:

```
Event control_name.event_name
        ... SCRIPT statements here ...
End Event
```

where *control_name* is the name of the control, and *event_name* is the event handler name itself. For example, the following code snippet illustrates the use of declaration, definition and the event handler:

```
<ELS_QPARAMS>
    QUERY_OPTIONS
    [
            UseShipDate    CHECKBOX_CONTROL,
            ShipDate       DATE CONTROL
    ];
```

```
    UseShipDate.Name = "Use ShipDate:";
    ShipDate.Name = "Ship Date:";
    ShipDate.Enabled = FALSE;

    Event UseShipDate.OnChange
           IF UseShipDate.GetValue() = 1 THEN
                  ShipDate.Enabled = TRUE;
           ELSE
                  ShipDate.Enabled = FALSE;
           END IF
    End Event

</ELS_QPARAMS>
```

In addition to properties and events, all controls have the following methods:

`GetSQLValue()` this function returns an SQL substring based on the `ValueID` property of the control, which at run-time can be used in the *WHERE*-clause of an SQL query,

`GetOperator()` this function returns the operator that is selected by the user in the operator combo-box.

We now proceed on listing other properties that are specific to each type of control. First, we note that the `STATIC_CONTROL` control has no additional properties, while the controls `CHECKBOX_CONTROL`, `OPTION_CONTROL` and `DATE_CONTROL`, have the following two additional properties or methods:

Property or Method	Description
`Default`	This property defines the default value of the control.
`ValueID`	This property defines the name of the row-identifier column, which is used in the string value returned by the *GetSQLValue()* function.
`GetValue()`	This function returns the control's user-entered value at run-time.

Table 2.13. *Additional properties and methods*

We consider next the `TEXTBOX_CONTROL`, which in addition to the properties, methods and event handlers of tables 2.12 and 2.13, has the `Mask` property. This property essentially is a string defining the input mask for the textbox control, for example, the following code snippet will allow entry of only phone numbers masked in (999) 999-9999 form:

```
    QUERY_OPTIONS
    [
           PhoneNum TEXTBOX CONTROL     // control variable declaration
    ];
    PhoneNum.Name = "Phone: ";          // the caption of the control
    PhoneNum.Width = 80;
    PhoneNum.Mask = "(###) ###-####";   // input mask (999) 999-9999
```

For the definition of input mask the following mask specification symbols may be used:

Mask Symbol	Description
`&`	This input mask will allow an ASCII character, for example "`&&&&&`" allows the following input values: `a_8!*`, `8F^w1`
`A`	This input mask will allow an alphanumeric character, i.e. a-z, A-Z, 0-9 characters, for example "`AAAAAA`" allows the following input values: `ab78`, `83200F`
`?`	This input mask will allow a character from a-z or A-Z ranges, for example "`????`" allows the following input values: `ab`, `FJKe`
`#`	This input mask will allow a digit 0-9, for example "`###-####`" allows the following input values: `578-9057`,

	832-00
U	This input mask will allow a character from a-z or A-Z ranges, and will force to upper case
L	This input mask will allow a character from a-z or A-Z ranges, and will force to lower case
\	This symbol is the escape character which may be used to interpret the following mask symbol as the actual literal character. For example, "??\###" which allows the following value MI#67

Table 2.14. *Mask specification symbols with the corresponding interpretation*

Note that by default the value of the Mask property is the empty string, which will allow the input of arbitrary size strings comprising of ASCII characters.

We consider next the SPIN_CONTROL, which in addition to the properties, methods and event handlers of tables 2.12 and 2.13, has the following two properties:

Property or Method	Description
LBound	This property defines the smallest integer value that can be scrolled via the lower spin arrow
UBound	This property defines the largest integer value that can be scrolled via the upper spin arrow

Table 2.15. *Properties specific to the SPIN_CONTROL*

Along with the common properties, methods and event handlers, the LISTBOX_CONTROL has the following more specific properties or methods:

Property or Method	Description
Source	This property defines the content of the list-box control. There are two ways to evaluate this property, setting it either to a VARCHAR array variable or to a DATASOURCE variable.
ValueID	This property defines the name of the row-identifier column, which is used in the string value returned by the *GetSQLValue()* function. In the case when the Source is defined via a DATASOURCE variable, this identifier column must be a fieldname included in the field list of the recordset corresponding to the DATASOURCE variable.
ShowValueID	This property defines whether to show or hide the ValueID column in the displayed list. Note that this applies only when the Source is defined via a DATASOURCE variable.
SelCount	This property returns the number of items in the list-box selected by the user at run-time. Note that this property is read-only.
GetValue(nSel)	This function returns the value of the nSel selected item, from the items selected from the list-box by the user (at run-time). Note that nSel is between 1 and SelCount. We emphasize that when the Source is defined via a DATASOURCE variable, the returned value corresponds to the row-identifier column determined by ValueID. Otherwise when the Source is defined via VARCHAR array variable, the returned value is the value of the item.
AllItems	This property defines the state of the **All Items** check-box that accompanies a LISTBOX_CONTROL.

Table 2.16. *Properties specific to the LISTBOX_CONTROL*

We illustrate the use of these properties in the following code segment, in which an array variable is used to define the content of the list-box:

```
QUERY OPTIONS
[
       lst LISTBOX_CONTROL   // control variable declaration
];

// declare and set an array
DECLARE @SCntry(5) VARCHAR(10);
SET @SCntry(0) = "France";
```

```
SET @SCntry(1) = "Germany";
SET @SCntry(2) = "USA";
SET @SCntry(3) = "Mexico";
SET @SCntry(4) = "Brazil";

lst.Name = "Ship Country: ";        // the caption of the control
lst.ValueID = "ShipCountry";        // identify user-selected values as ShipCountry field
lst.Source = @SCntry(5);            // set the source of the list-box
```

We will also illustrate, later in this section, the usage of the LISTBOX_CONTROL, in the case when the Source is defined by a data source instead of array variable.

The COMBOBOX_CONTROL along with the common properties, methods and event handlers, has the following more specific properties or methods:

Property or Method	Description
Source	This property defines the content of the list inside the combo-box control. There are two ways to evaluate this property, setting it either to a VARCHAR array variable or to a DATASOURCE variable.
ValueID	This property defines the name of the row-identifier column, which is used in the string value returned by the *GetSQLValue()* function. In the case when the Source is defined via a DATASOURCE variable, this identifier column must be a fieldname included in the field list of the recordset corresponding to the DATASOURCE variable.
ShowValueID	This property defines whether to show or hide the ValueID column in the displayed list. Note that this applies only when the Source is defined via a DATASOURCE variable.
Default	This property defines the default value of the control.
GetValue()	This function returns the value of the item selected by the user (at run-time). We emphasize that when the Source is defined via a DATASOURCE variable, the returned value corresponds to the row-identifier column determined by ValueID. Otherwise when the Source is defined via VARCHAR array variable, the returned value is the value of the item.
EditField	This property defines the name of the field that has its corresponding selected value displayed in the edit-box of the combo-box. Note that this property applies only in the case when the Source is defined via a DATASOURCE variable.

Table 2.17. *Properties specific to the COMBOBOX_CONTROL*

The following code snippet illustrates the use of the COMBOBOX_CONTROL:

```
QUERY OPTIONS
[
       combo   COMBOBOX_CONTROL
];

combo.Name = "Orders: ";
combo.Source = @MyOrders;      // where @MyOrders is a datasource of Orders table
combo.EditField = "ShipName"; // the value of ShipName field will show in the edit-box
combo.ValueID = "OrderID";     // the key-field is OrderID
combo.ShowValueID = FALSE;     // do not show the key-field in drop-down list
```

The DATERANGE_CONTROL has the following additional properties and methods along with the common ones

Property or Method	Description
ValueID	This property defines the name of the row-identifier column, which is used in the string value returned by the *GetSQLValue()* function.
StartDefault	This property defines the default value of the start control of the range.
EndDefault	This property defines the default value of the end control of the range.

GetStartValue()	This function returns the value of the start control entered by the user (at run-time).
GetEndValue()	This function returns the value of the end control entered by the user (at run-time).

Table 2.18. *Properties specific to the DATERANGE_CONTROL*

The following code snippet illustrates the usage of the DATERANGE_CONTROL:

```
QUERY_OPTIONS
[
        drng DATERANGE CONTROL
];

drng.Name = "Sales Period: ";
drng.StartDefault = DateAdd(MM, -2, GetDate());
drng.EndDefault = DateAdd(MM, 2, GetDate());
```

and at some other location in the report script, outside the *ELS-QPARAMS* section, the start and end values entered by the user (at run-time) may be called, for example, in the following manner, where `[Employee Sales by Country]` is a stored procedure in the *NorthWind* database of the MS-SQL Server :

```
DECLARE @sSP VARCHAR(100);

SET @sSP = "exec [Employee Sales by Country] '" +
              Format(drng.GetStartValue(), "mm/dd/yyyy") + "', '" +
              Format(drng.GetEndValue(), "mm/dd/yyyy") + "'";
```

The NUMBERRANGE_CONTROL has the following additional properties and methods along with the common ones

Property or Method	Description
ValueID	This property defines the name of the row-identifier column, which is used in the string value returned by the *GetSQLValue()* function.
StartDefault	This property defines the default value of the start control of the range.
EndDefault	This property defines the default value of the end control of the range.
StartMask	This property defines the input mask for the start control of the range.
EndMask	This property defines the input mask for the end control of the range.
GetStartValue()	This function returns the value of the start control entered by the user (at run-time).
GetEndValue()	This function returns the value of the end control entered by the user (at run-time).

Table 2.19. *Properties specific to the NUMBERRANGE_CONTROL*

The following code snippet illustrates the usage of the NUMBERRANGE_CONTROL:

```
QUERY_OPTIONS
[
        nrng NUMBERRANGE_CONTROL
];

nrng.Name = "ShipVia Range: ";
nrng.StartDefault = @nNum;
nrng.EndDefault = @nNum + 10;
nrng.StartMask = "###";
nrng.EndMask = "###";
```

We consider now the GRID_CONTROL, which has the following additional properties and methods along with the

common ones:

Property or Method	Description
ValueID(nCol)	This property defines the name of the row-identifier column corresponding to the specified grid-column, this identifier column is used in the string value returned by the *GetSQLValue()* function. The argument nCol must be between 1 and ColumnCount.
Mask(nCol)	This property defines the input mask for the specified grid-column. The argument nCol must be between 1 and ColumnCount.
ColumnCount	This property defines the number of the columns for the grid control.
RowCount	This property returns the number of rows for the grid control. Note that this property is read-only and depends on the number of rows entered by the user at run-time.
GetValue(nRow, nCol)	This function returns the value of the grid-cell specified by nRow and nCol.

Table 2.20. *Properties specific to the GRID_CONTROL*

Finally we have the GROUP and SEPARATE BY controls which are essentially used to group controls or add specific space between controls. Grouping is defined in the QUERY_OPTIONS via the following generic syntax:

```
GROUP group_name
{
        control_name_1 control_type_1,
        control_name_2 control_type_2,
        ...
        control_name_N control_type_N
}
```

The spacer control syntax is as follows:

```
SEPARATE BY pixel_height
```

The following sample code illustrates the usage of both group and spacer controls, as well as, usage of event handlers and other features:

```
QUERY_OPTIONS
[
        GROUP "Search"
        {
                ByShipDate      OPTION_CONTROL,
                ByDateRange     OPTION_CONTROL,
                ByShipVia       OPTION_CONTROL
        },
        SEPARATE BY 10,
        ShipDate        DATE_CONTROL,
        drng            DATERANGE_CONTROL,
        shpvrng         NUMBERRANGE_CONTROL
];
ByShipDate.Name = "By Ship Date";
ByDateRange.Name = "By Date Range";
ByShipVia.Name = "By Ship Via Range";
ByShipDate.Default = TRUE;

drng.Enabled = FALSE;
drng.Name = "Ship Date Range:";
ShipDate.Name = "Ship Date:";
ShipDate.Default = GetDate();
shpvrng.Name = "Ship Via Range:";
shpvrng.Enabled = FALSE;
shpvrng.StartMask = "###";
shpvrng.EndMask = "###";

Event ByShipDate.OnChange
        IF ByShipDate.GetValue() = 1 THEN
                ShipDate.Enabled = TRUE;
                drng.Enabled = FALSE;
                shpvrng.Enabled = FALSE;
```

```
        ELSE
                ShipDate.Enabled = FALSE;
        END IF
End Event

Event ByDateRange.OnChange
        IF ByDateRange.GetValue() = 1 THEN
                drng.Enabled = TRUE;
                ShipDate.Enabled = FALSE;
                shpvrng.Enabled = FALSE;
        ELSE
                drng.Enabled = FALSE;
        END IF
End Event

Event ByShipVia.OnChange
        IF ByShipVia.GetValue() = 1 THEN
                drng.Enabled = FALSE;
                ShipDate.Enabled = FALSE;
                shpvrng.Enabled = TRUE;
        ELSE
                shpvrng.Enabled = FALSE;
        END IF
End Event
```

Along with the query option controls there are the following list item controls:

LIST_ITEM.CHECK	check-box list-item control corresponding to the CHECKBOX_CONTROL,
LIST_ITEM.OPTION	option-button list-item control corresponding to the OPTION_CONTROL,
LIST_ITEM.TEXT	textbox list-item control corresponding to the TEXTBOX_CONTROL,
LIST_ITEM.STATIC	static list-item control, corresponding to the STATIC_CONTROL,
LIST_ITEM.COMBO	combo-box list-item control corresponding to the COMBOBOX_CONTROL,
LIST_ITEM.LIST	list-box list-item control corresponding to the LISTBOX_CONTROL,
LIST_ITEM.SPIN	spin list-item control corresponding to the SPIN_CONTROL,
LIST_ITEM.DATE	date list-item control corresponding to the DATE_CONTROL,
LIST_ITEM.DATERANGE	date-range list-item control corresponding to the DATERANGE_CONTROL,
LIST_ITEM.NUMBERRANGE	number-range list-item control corresponding to the NUMBERRANGE_CONTROL.

These controls essentially are the same as the corresponding query option controls (on the right side), except that they are initially hidden and only the names appear in a special list-box. Such a control becomes visible only when the user selects the name of the control in the special list-box. These types of controls are recommended, specially, when one wants to use a lot of controls in the **Query Form** section of the **Report Generator** window and is running out of window or screen space.

Along with direct scripting, query forms may be constructed via the **Query Form** designer sliding window. This window can be called via the **Query Form** menu item under the **Tools** menu, or alternatively via the **Query Form** toolbar button of the **Project Explorer** pane.

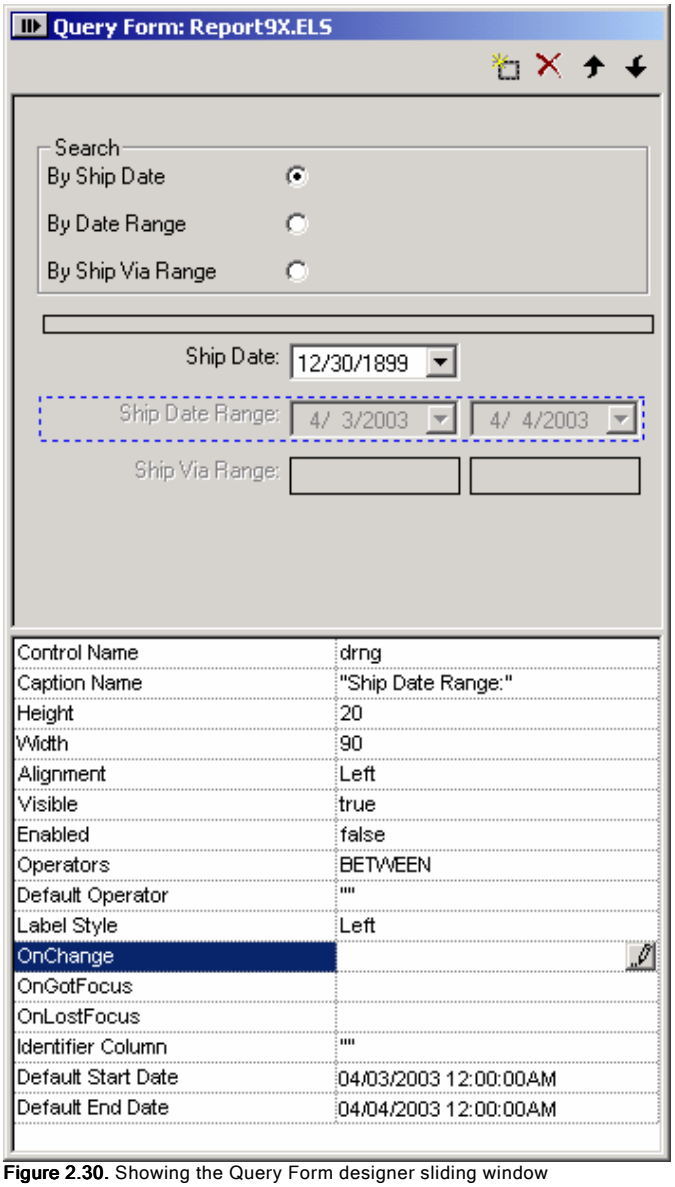


Figure 2.30. Showing the Query Form designer sliding window

The **Query Form** designer window essentially consists of the following sections: *Toolbar*, *Control List* and *Properties* panes (see Figure 2.30 for more details).

The *Toolbar* contains the following command buttons:

Add Control	to add a new control,
Delete Control	to delete selected control,
Move Up	to move the selected control one step up,
Move Down	to move the selected control one step down.

Note that the **Add Control** button will call the **Add Control** dialog, via which a query option control may be selected (see Figure 2.31 for more details).

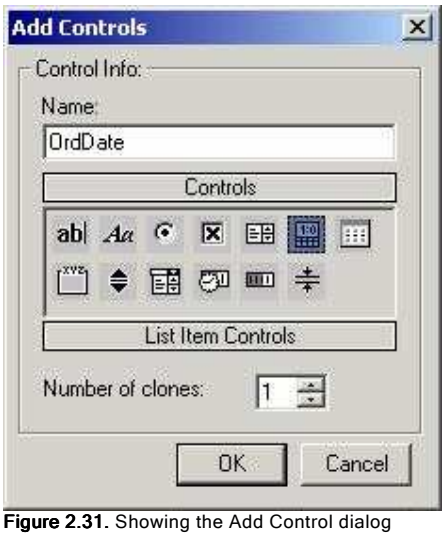


Figure 2.31. Showing the Add Control dialog

In the **Query Form** window, a horizontal splitter control divides the window into two panes. The top pane being the *Control List*, is essentially a list in which controls are added, deleted or moved up or down.The bottom pane is the *Properties* pane, which displays the properties of the control currently selected in the *Control List* pane.

Observe that since the query form section in the **Report Generator** window inherently has a narrow screen space, the **Query Form** designer is intentionally designed to arrange controls aligning them into a vertical list.
To set or define the properties of a control in the *Control List*, one may simply select the control and edit the desired properties in the *Properties* pane. Note that some properties are editable directly in the *Properties* pane, while others are not and must be manually edited in the **Source** view, in which case a double-click on the property name (i.e. left column cell) provides the jump to the location in the script where the property is to be set. We should emphasize that some properties when defined via expressions in the **Source** view may no longer be editable directly in the *Properties* pane, and therefore the double-click technique may be used to jump to the specific code location in the script. Also, observe that the value cell (i.e. right column cell) of the event properties *OnChange*, *OnGotFocus* and *OnLostFocus* have small buttons, which may be used to jump to the code location via a single mouse-click, as an alternative method to the double-click on the property name.

In the rest of this section we will present a sample report, which illustrates the use of query options with events, as well as other related issues. First, prepare an empty report with the name REPORT8.ELS. Then create three query objects in the **AUX\Queries** container node of the MyNWDB connection. The first query, Categories, must have the following SQL command:

```
SELECT
        CategoryName,
        CategoryID
FROM
        Categories
```

The second query, Products, must have the following SQL command:

```
SELECT
        ProductName,
        ProductID
FROM
        Products
WHERE
        (CategoryID = 1)
ORDER BY
        ProductName
```

These two queries will be used in the query form section of the **Report Generator** window, the third query, ProdOrders, must be used in the report output and must have the following SQL command:

```
SELECT
        Orders.OrderID,
        Orders.OrderDate,
        Orders.CustomerID,
        Orders.ShipCity,
        [Order Details].Quantity,
        [Order Details].UnitPrice
FROM
        Orders
         INNER JOIN [Order Details] ON
        [Order Details].OrderID = Orders.OrderID
WHERE
        ([Order Details].ProductID = 1)
```

In this way, we want to develop a report which will generate all the orders that contain a particular product selected in the query form section of the report.
To setup the query form, use the **Query Form** designer to add two combo-box controls in the *ELS_QPARAMS* report section, namely:

`Category` `COMBOBOX_CONTROL` to contain all the categories from the Categories query,
`Product` `COMBOBOX_CONTROL` to contain all the products belonging to the category selected in the `Category` combo-box.

Then drag-drop the Categories query into the script and setup the properties of the two controls so that we will have the following code inside the *ELS_QPARAMS* section:

```
DECLARE @Categories DATASOURCE;

SET @Categories = "SELECT " +
                        "CategoryName, " +
                        "CategoryID " +
                  "FROM " +
                        "Categories";

QUERY_OPTIONS
[
    Category COMBOBOX_CONTROL,
    Product  COMBOBOX CONTROL
];

Category.Name = "Product Category: ";
Category.ValueID = "CategoryID";         // ID field is CategoryID
Category.EditField = "CategoryName";     // field displayed in the edit-box of the combobox
Category.ShowValueID = FALSE;            // and do not show this ID field
Category.Source = @Categories;

Product.Name = "Product: ";
Product.ValueID = "ProductID";           // ID field is ProductID
```

```
Product.EditField = "ProductName";       // field displayed in the edit-box of the combobox
Product.ShowValueID = FALSE;             // and do not shoe this ID field
```

Note that the identifier field for the source of the `Category` control is *CategoryID* field, while the value of the *CategoryName* field is displayed in the edit-box of the combo-box. Similarly, for the `Product` control the identifier field is *ProductID*. Also observe that the identifier fields for both controls are hidden in their respective drop-down lists. Now when the user selects an item from the `Category` combo-box control, the returned value is the value corresponding to the *CategoryID* identifier field. This value will be used to populate the `Product` combo-box control with the list of products belonging to the selected category.

Next, we utilize the **Query Form** designer and select the `Category` control in the control list, and click on the edit button of the *OnChange* event handler. This will insert the corresponding event handler section and jump to the location in the report script. Now drag-drop the Products query into this event handler section, and set the `Source` property of the `Product` control to the `@Products` data source variable, so that the following addition code will be added in the *ELS_QPARAMS* report section:

```
Event Category.OnChange
     DECLARE @Products DATASOURCE;
     SET @Products = "SELECT " +
                             "ProductID, " +
                             "ProductName " +
                     "FROM " +
                             "Products " +
                     "WHERE " +
                             "(CategoryID = " + Category.GetValue() + ") " +
                     "ORDER BY " +
                             "ProductName";
     Product.Source = @Products;
End Event
```

Note that this event handler will trigger the redefinition of the `@Products` data source whenever the combo-box selection is changed. Looking at the SQL command, observe that whenever the value of the `Category` combo-box is changed, the `GetValue()` function will return the *CategoryID* value, and therefore the `@Products` data source will be filtered according to this value.

Continuing the construction of the rest of the report, concentrate on the *ELS_RDETAIL* report section and add a pure *ELS* section by inserting *ELS*-tags. Then drag-drop the ProdOrders query into this section, and modify the *WHERE*-clause so that the *ProductID* field is set to the value returned by the `Product` combo-box control. The result is as follows:

```
<ELS_RDETAIL FONT-FAMILY="Times New Roman" FONT-SIZE="8pt">
<ELS>
DECLARE @ProdOrders DATASOURCE;
SET @ProdOrders = "SELECT " +
            "Orders.OrderID, " +
            "Orders.OrderDate, " +
            "Orders.CustomerID, " +
            "Orders.ShipCity, " +
            "[Order Details].Quantity, " +
            "[Order Details].UnitPrice " +
     "FROM " +
            "Orders " +
             "INNER JOIN [Order Details] ON " +
                   "[Order Details].OrderID = Orders.OrderID " +
     "WHERE " +
            "([Order Details].ProductID = " + Product.GetValue() + ")";
</ELS>

</ELS_RDETAIL>
```

Using the **Insert ELS-Row** window insert two *ELS-Rows* into the *ELS-RDETAIL* report section, one for main ProdOrders records and the other for totals. Then utilizing the **Data Fields** window make field insertions at

respective columns of the main *ELS-Row*, so that after some manual modifications the report detail section will have the following code:

```
<ELS_RDETAIL FONT-FAMILY="Times New Roman" FONT-SIZE="8pt">
<ELS>
DECLARE @ProdOrders DATASOURCE;
SET @ProdOrders = "SELECT " +
            "Orders.OrderID, " +
            "Orders.OrderDate, " +
            "Orders.CustomerID, " +
            "Orders.ShipCity, " +
            "[Order Details].Quantity, " +
            "[Order Details].UnitPrice " +
    "FROM " +
            "Orders " +
             "INNER JOIN [Order Details] ON " +
             "[Order Details].OrderID = Orders.OrderID " +
    "WHERE " +
            "([Order Details].ProductID = " + Product.GetValue() + ")";

DECLARE @nCount, @nQty int;      // variables used for totals
DECLARE @mTotal money;

</ELS>
// main ProdOrders ELS-Row
<ELS ROW NAME="ELSRow1">
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
            <C HEIGHT="15" WIDTH="9.58%">
                   <HDR><B>OrderID</B></HDR>
                   <FLD>Format(@ProdOrders.Column("OrderID"), "")</FLD>
            </C>
            <C HEIGHT="15" WIDTH="13.75%">
                   <HDR><B>Order Date</B></HDR>
                   <FLD>Format(@ProdOrders.Column("OrderDate"), "mm/dd/yyyy")</FLD>
            </C>
            <C HEIGHT="15" WIDTH="11.67%">
                   <HDR><B>CustomerID</B></HDR>
                   <FLD>@ProdOrders.Column("CustomerID")</FLD>
            </C>
            <C HEIGHT="15" WIDTH="42.08%">
                   <HDR><B>Ship City</B></HDR>
                   <FLD>@ProdOrders.Column("ShipCity")</FLD>
            </C>
            <C ALIGN="right" HEIGHT="15" WIDTH="10.00%">
                   <HDR><B>Quantity</B></HDR>
                   <FLD>Format(@ProdOrders.Column("Quantity"), "###")</FLD>
            </C>
            <C ALIGN="right" HEIGHT="15" WIDTH="12.92%">
                   <HDR><B>UnitPrice</B></HDR>
                   <FLD>Format(@ProdOrders.Column("UnitPrice"), "####.00")</FLD>
            </C>
    </L>
</ELS_ROW>
// ELS-Row used for totals
<ELS ROW NAME="ELSRow2">
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
            <C HEIGHT="15" WIDTH="67.08%">
                   <B><FLD>@nCount</FLD> RECORDS</B>
            </C>
            <C ALIGN="right" HEIGHT="15" WIDTH="10.00%">
                   <B>TOTALS: </B>
            </C>
            <C ALIGN="right" BGCOLOR="#F1F1F1" HEIGHT="15" WIDTH="10.00%">
                   <B><FLD>Format(@nQty, "#######")</FLD></B>
            </C>
            <C ALIGN="right" BGCOLOR="#F1F1F1" HEIGHT="15" STYLE="BORDER-LEFT: #ffffff 3pt solid"
                                                                  WIDTH="12.92%">
                   <B><FLD>Format(@mTotal, "########.00")</FLD> </B>
            </C>
    </L>
</ELS_ROW>

<ELS>
```

```
SET @nCount = 0;
SET @nQty = 0;
SET @mTotal = 0.00;

BeginHeader("ELSRow1");
WHILE NOT @ProdOrders.Eof()
     ResultRow("ELSRow1");
     SET @nQty = @nQty + @ProdOrders.Column("Quantity");
     SET @mTotal = @mTotal + @ProdOrders.Column("UnitPrice");
     @ProdOrders.Next();
     SET @nCount = @nCount + 1;
END LOOP
EndHeader("ELSRow1");
// write totals
ResultRow("ELSRow2");
</ELS>
</ELS_RDETAIL>
```

Note that at the beginning of the *ELS_RDETAIL* section we have declared variables `@nCount`, `@nQty` and `@mTotal`, to keep track of respectively the totals of records, quantities and prices. Observe that these variables are initialized to zero before the iteration process starts over the `@ProdOrders` data source, and during the iteration these variables are incremented by the respective values of the data fields. Finally, the row-specification "`ELSRow2`" is used to present these totals at the end of all `@ProdOrders` records.

Pending Variables

In this section we will cover the concept of variable evaluation and output with respect to its logical position in the report script. More precisely, the problem that we will address has to do with the physical position of the output of an expression value in the report output, verses the logical position of the evaluation of this expression in the report script. If the particular variable's value is output after this variable's evaluation, then we call this variable a *running* variable, in a sense, because the variable's evaluation runs along the flow of the report generation process. For example, totals and summaries at the end of a report may be implemented via *running* variables, which may be incremented as running totals along the logical flow of the report generation. In contrast, a *pending* variable is a variable that has evaluation position after the physical output location of the report script, and therefore the writing or outputting of the value of the variable must wait until proper evaluation process takes place during the report generation. Pending variables are necessary, for example, when totals or summaries in a report are presented at the beginning of the report rather than at the end.

In general, any variable of basic data type may be made into pending variable by simply applying the *ValueAt* function. Recall that the *ValueAt* function has the following syntax:

```
VALUEAT(Variable, Event)
```

Where *`Variable`* is any variable of basic data type, and *`Event`* is one of the following report engine events:

```
ELS_OnBeginPage,
ELS_OnEndPage,
ELS_OnBeginReport,
ELS_OnEndReport.
```

Note that the *`Variable`* argument must be the name of the variable and not an expression. If we want to pass the value of an expression, then we must first assign the expression to a variable and pass the variable's name as argument to this function.

To illustrate the usage of pending variables we will outline the sample report REPORT10.ELS, which is a report that has a summary row at the beginning of the report. Moreover, this summary row contains totals that depend on the detail records that follow it. In the *ELS-QPARAMS* section of this new report we define the `Prod` parameter option using a data source reference to the Products table of the *Northwind* database, as shown below:

```
<ELS_QPARAMS>
     PARAM OPTIONS
     [
```

```
            Prod int
    ];

    DECLARE @Product DATASOURCE;
    SET @Product = "SELECT " +
                        "ProductName, " +
                        "ProductID " +
                "FROM " +
                        "Products";

    Prod.Prompt = "Product: ";
    Prod.SourceType = 1;
    Prod.ValidValues = (@Product, "ProductName", "ProductID");
    Prod.DefaultValues = (@Product, "ProductName", "ProductID");

</ELS_QPARAMS>
```

The report data will be defined over the Invoices view object with the corresponding `@Orders` data source defined in the *ELS_RSETTINGS* below:

```
<ELS_RSETTINGS>
    SET REPORT_TITLE      = "Pending Variable Report";
    SET PAGE ORIENTATION  = ELS PORTRAIT;
    SET PAGE_SIZE         = ELS_LETTER;
    SET PAGE SOURCE       = ELS UPPER;
    SET PAGE_MARGINS.LEFT        = 0.5;
    SET PAGE_MARGINS.RIGHT       = 0.5;
    SET PAGE_MARGINS.TOP         = 0.75;
    SET PAGE_MARGINS.BOTTOM      = 0.5;
    SET DEFAULTMODE              = ELS FAST;

    // reports data per selected ProductID sorted by date
    DECLARE @Orders DATASOURCE;
    SET @Orders = "SELECT " +
                        "OrderID, " +
                        "OrderDate, " +
                        "CustomerName, " +
                        "ShipCity, " +
                        "Quantity, " +
                        "UnitPrice, " +
                        "ExtendedPrice " +
                "FROM " +
                        "Invoices " +
                "WHERE productid = " + Format(Prod.Value, "") +
                " ORDER BY Invoices.OrderDate";

    // get product's name
    DECLARE @sProd varchar(50);
    SET @sProd = @Orders.Column("ProductName");

    // variables used in the summary
    DECLARE @nCount, @nQty int;
    DECLARE @mTotal money;
    DECLARE @sTotalQty, @sTotalSale VARCHAR(20);
</ELS RSETTINGS>
```

The variables `@sTotalQty` and `@sTotalSale` along with the `@nCount` variable will be the pending variables for this report sample. In particular, they are made pending via the *ValueAt* function applied on the totals in the "`ELSRow2`" row-specification, as is shown below in the *ELS_RDETAIL* section:

```
<ELS_RDETAIL FONT-FAMILY="Times New Roman" FONT-SIZE="10pt">
<ELS_ROW NAME="ELSRow1">
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
        <C WIDTH="10%" HEIGHT="15">
            <HDR><B>OrderID</B></HDR>
             <FLD>Format(@Orders.Column("OrderID"), "")</FLD>
        </C>
        <C WIDTH="13%" HEIGHT="15">
            <HDR><B>Order Date</B></HDR>
```

```
             <FLD>Format(@Orders.Column("OrderDate"), "mm/dd/yyyy")</FLD>
        </C>
        <C WIDTH="28%" HEIGHT="15">
            <HDR><B>Customer</B></HDR>
             <FLD>@Orders.Column("CustomerName")</FLD>
        </C>
        <C WIDTH="27%" HEIGHT="15">
            <HDR><B>Ship City</B></HDR>
             <FLD>@Orders.Column("ShipCity")</FLD>
        </C>
        <C ALIGN="right" WIDTH="10%" HEIGHT="15">
            <HDR><B>Quantity</B></HDR>
             <FLD>Format(@Orders.Column("Quantity"), "#,##0")</FLD>
        </C>
        <C ALIGN="right" WIDTH="12%" HEIGHT="15">
            <HDR><B>Subtotal</B></HDR>
             <FLD>Format(@Orders.Column("ExtendedPrice"), "$#,##0.00")</FLD>
        </C>
    </L>
</ELS_ROW>

// this row is used for pending variables
<ELS ROW NAME="ELSRow2">
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
        <C HEIGHT="15" WIDTH="68%">
            <B><FLD>ValueAt(@nCount, ELS_OnEndReport)</FLD></B> records of
                                <B><FLD>@sProd</FLD></B>
        </C>
        <C HEIGHT="15" ALIGN="right" WIDTH="10.00%">
            <B>TOTALS: </B>
        </C>
        <C HEIGHT="15" ALIGN="right" BGCOLOR="yellow" WIDTH="10%">
            <B><FLD>ValueAt(@sTotalQty, ELS_OnEndReport)</FLD></B>
        </C>
        <C HEIGHT="15" ALIGN="right" BGCOLOR="yellow"
                        STYLE="BORDER-LEFT: #ffffff 3pt solid" WIDTH="12%">
            <B><FLD>ValueAt(@sTotalSale, ELS OnEndReport)</FLD> </B>
        </C>
    </L>
</ELS_ROW>

<ELS>
SET @nCount = 0;
SET @nQty = 0;
SET @mTotal = 0.00;

// write totals at the front
ResultRow("ELSRow2");
</ELS>
<BR>
<ELS>
// write details next
BeginHeader("ELSRow1");
WHILE NOT @Orders.Eof()
    ResultRow("ELSRow1");
    SET @nQty = @nQty + @Orders.Column("Quantity");
    SET @mTotal = @mTotal + @Orders.Column("ExtendedPrice");
    @Orders.Next();
    SET @nCount = @nCount + 1;
END LOOP
EndHeader("ELSRow1");

// convert totals into string
SET @sTotalQty = Format(@nQty, "#,##0");
SET @sTotalSale = Format(@mTotal, "$#,##0.00");
</ELS>
</ELS_RDETAIL>
```

According to the application of the *ValueAt* function, the evaluation of the `@nCount`, `@sTotalQty` and `@sTotalSale` variables must wait until the end of the report event is fired by the report generation process. Therefore, it is by virtue of these pending mechanisms that totals of detail records may be put at a position in the output prior to the detail records themselves, as is depicted in the generation code at the botton of the *ELS_RDETAIL* section.

This completes the illustration of the usage of pending variables, in future chapters we will give more elaborate use of pending variables in much more complex grouping and summaries.

Report Settings

In this section we outline a description of the report settings parameters and the corresponding effect on the page settings of the report output. In general, the report setting must be specified, especially for paginated report output, and essentially consists of the following parameters:

`REPORT_TITLE` This parameter specifies the internal report title, which becomes the HTML document title, and therefore may be used by HTML agents. When used in a report that is furnished with a query form, this title string is displayed as the description of the report in the **Report Generator** window.

`PAGEMARGIN` This parameter specifies the page margins of the output pages in decimal inches, and has the following four properties:

- `TOP` which controls the top margin of the page,
- `BOTTOM` which controls the bottom margin of the page,
- `LEFT` which controls the left margin of the page,
- `RIGHT` which controls the right margin of the page.

`PAGE_SIZE` This parameter specifies a standard printer page size, by default it is *Letter* size.

`PAGE_SOURCE` This parameter specifies a standard printer source, by default it is *Upper* tray.

`PAGE_ORIENTATION` This parameter specifies a standard page orientation, possible page orientation values are `ELS_PORTRAIT` (default) and `ELS_LANDSCAPE`.

`DEFAULTMODE` This parameter specifies the report generation mode. There are three report generation modes, namely: `ELS_FAST` (default), `ELS_CONTINUOUS` and `ELS_STYLE`.

`SUPPRESS_PHEADER` This parameter specifies whether to include or exclude the page header section of the report for the first and last pages, and has the following properties:

- `FIRSTPAGE` which controls the inclusion/exclusion in the first page,
- `LASTPAGE` which controls the inclusion/exclusion in the last page.

`SUPPRESS_PFOOTER` This parameter specifies whether to include or exclude the page footer section of the report for the first and last pages, and has the following properties:

- `FIRSTPAGE` which controls the inclusion/exclusion in the first page,
- `LASTPAGE` which controls the inclusion/exclusion in the last page.

In the current version of the SCRIPT report engine, the `ELS_STYLE` generation mode is not very useful and is not recommended in the case of long reports. In contrast therefore, there are two useful modes of generating reports, namely, a *paginated* report output or a *continuous* report output. The `ELS_FAST` mode will generate reports in a paginated and extremely fast manner, whereas, the `ELS_CONTINUOUS` mode will generate reports in a continuous manner, which perhaps is more suitable to web-based reporting.

Finally, the user may set the report settings either directly entering information into the *ELS_RSETTINGS* report section, in which case *IntelliSence* makes life extremely easy by providing indispensable selection lists. Or the user may call the **Report Settings** dialog via the **Report Settings** menu item under the **File** menu. For further details about the **Report Settings** dialog see Figure 2.32.

Also, it is important to notice that the **List Report Settings** menu command in the popup menu triggered inside the **Source** view will popup an actual list of all the report settings primitives.

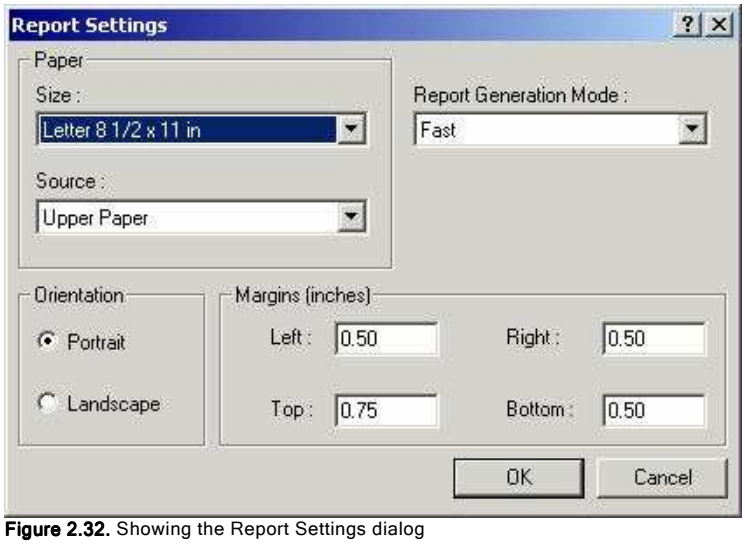


Figure 2.32. Showing the Report Settings dialog

XML Object

The XML object support was introduced directly into the SCRIPT language in version 3.0 of ELS-Script®. In particular, it implements all elements of MSXML 4.0 including full support for the *Document Object Model* (DOM), the *XML Schema Definition* language (XSD), the *Schema Object Model* (SOM), the *Extensible Stylesheet Language Transformation* (XSLT), *XML Path* language (XPath), and the *Simple API for XML* (SAX).

The following list outlines a brief description of all the XML object types that can be declared and defined directly in the SCRIPT language:

Object	Description
XMLDOM	This object represents the top level of the XML source, includes members for retrieving and creating all other XML objects
XMLDOMAttribute	This object represents an attribute of the XMLDOMElement object
XMLDOMCDATASection	This object is used to quote or escape blocks of text to keep from being interpreted as markup language
XMLDOMCharacterData	This object provides text manipulation methods that are used be several objects
XMLDOMComment	This object represents the content of an XML comment
XMLDOMDocumentFragment	This object is used for tree insert operations
XMLDOMDocumentType	This object contains information associated with the document type declaration
XMLDOMElement	This object represents the element object
XMLDOMEntity	This object represents a parsed or unparsed entity in the XML document
XMLDOMEntityReference	This object represents an entity reference node
XMLDOMNamedNodeMap	This object adds support for namespaces and iteration through the collection of attribute nodes
XMLDOMNode	This object represents the XML node and supports data types, namespaces, document type definition (DTD), and schemas
XMLDOMNodeList	This object represents the collection of nodes with indexed access
XMLDOMNotation	This object contains a notation declared in the document type definition (DTD) or schema
XMLDOMParseError	This object returns information about the last parse error, including the error number,

	line number, character position and description of error
XMLDOMProcessingInstruction	This object represents a processor-specific instruction,
XMLDOMSchemaCollection	This object represents the schema or namespace collection
XMLDOMSelection	This object represents the list of nodes that match a given XML Path language (XPath) expression
XMLDOMText	This object represents the text content of an element or attribute

Table 2.21. *A brief aoutline of the XML objects types supported in SCRIPT language*

For further syntax details about the member properties and methods of these XML objects, please be informed that the **Source** view comes with full *IntelliSense* support, making the XML utilization an extremely intuitive development task. Further details may also be found in the online help that accompanies the software.

For the rest of this section we illustrate via a simple report, the usage of the XMLDOM, XMLDOMNode and XMLDOMNode XML objects, along with their indispensable selectNodes, selectSingleNode, item, firstChild and nodeValue member properties and methods. To proceed, create an empty report script with the name Report11.ELS. This report will be based on the Orders.XML, an XML document located in the \Samples\Data subfolder of the ELS-Script® software's main folder. In this report, we first add the OrderID report parameter option via direct entry into the **Source** view or with the help of the **Parameter List** window's more visual approach. Then, in the report settings section we define the XML source document utilizing the load member function:

```
DECLARE @oXML XMLDOM;
@oXML.load("C:\ELSS\Samples\Data\Orders.xml");
```

In this same section, we declare XMLDOMNodeList and XMLDOMNode objects to handle the XML dataset and the corresponding field level nodes. Then we define the dataset via the *XPath* pattern "dataroot/Orders" and query condition depending on the OrderID parameter option, as shown in the code segment below:

```
DECLARE @i INT;
DECLARE @oRecords XMLDOMNodeList;
DECLARE @oRec XMLDOMNode;
DECLARE @oFld(5) XMLDOMNode;

SET @oRecords = @oXML.selectNodes("dataroot/Orders[(OrderID >= "
                                          + Format(OrderID, "") + ")]");
```

Note that the @oRecords contains the collection of all nodes that have "Orders" as tag name. Therefore, we need to iterate over this collection and consider the field child nodes of each item. This is depicted in the code segment below:

```
WHILE @i < @oRecords.length
    SET @oRec = @oRecords.item(@i);

    SET @oFld(0) = @oRec.selectSingleNode("OrderID");
    SET @oFld(1) = @oRec.selectSingleNode("OrderDate");
    SET @oFld(2) = @oRec.selectSingleNode("ShipVia");
    SET @oFld(3) = @oRec.selectSingleNode("ShipName");
    SET @oFld(4) = @oRec.selectSingleNode("ShipCity");

    ResultRow("ELSRow1");

    SET @i = @i + 1;
END LOOP
```

Note that in particular, we are selecting via the selectSingleNode function the desired field node into the respective @oFld(j) XMLDOMNode object, while in the "ELSRow1" row-specification we will use the nodeValue of the firstChild of each @oFld(j) node to get the corresponding desired value for the field, as shown in the code segment below:

```
<ELS_ROW NAME="ELSRow1">
    ... column header line here ...
    <L border="0" CellSpacing="0" CellPadding="0" HEIGHT="15" WIDTH="100%">
          <C HEIGHT="15" WIDTH="10%">
```

```
                <FLD>
                COALESCE(@oFld(0).firstChild.nodeValue, "")
                </FLD>
        </C>
        <C ALIGN="center" HEIGHT="15" WIDTH="20%">
                <FLD>
                COALESCE(@oFld(1).firstChild.nodeValue, "")
                </FLD>
        </C>
        <C ALIGN="center" HEIGHT="15" WIDTH="10%">
                <FLD>
                COALESCE(@oFld(2).firstChild.nodeValue, "")
                </FLD>
        </C>
        <C HEIGHT="15" WIDTH="30%">
                <FLD>
                COALESCE(@oFld(3).firstChild.nodeValue, "")
                </FLD>
        </C>
        <C HEIGHT="15" WIDTH="30%">
                <FLD>
                COALESCE(@oFld(4).firstChild.nodeValue, "")
                </FLD>
        </C>
    </L>
</ELS_ROW>
```

Observe that we have used `COALESCE` function to handle the possibility when the named tag node may be missing for some *Orders* records. For example, it is possible that some values of some fields are *NULL* for some records, in which case, by XML conventions the whole XML tag will be omitted from the XML document. In such situations, the `COALESCE` function will simply insert the empty string value for the *FLD*-element instead of generating a run-time error.

Observe also that the `@oFld(1)` node contains the *OrderDate* date-time field value, but in the XML document (assuming that we have no accompanying XSD schema definition file), this information is presented in character data type. In SCRIPT language one may convert the character data to date-time and use the *Format* function to present the date value in a proper manner. This conversion may be achieved via the *ToDATE* function as depicted in the code segment below:

```
    Format( ToDATE( COALESCE(@oFld(1).firstChild.nodeValue, ""), "yyyy-mm-ddThh:nn:ss" ),
            "mm-dd-yyyy" )
```

Note that the *ToDATE* date-time specification "`yyyy-mm-ddThh:nn:ss`" was used based on the values of the *OrderDate* field node in the XML document. It may be interesting to note that such information may be derived programmatically from an accompanying XSD schema definition file, provided such a file indeed exists.

We close this section by listing the complete SCRIPT code for the Report11.ELS:

```
<ELS_QPARAMS>
    PARAM_OPTIONS
    [
            OrderID int
    ];
</ELS_QPARAMS>

<HTML>
<HEAD>
</HEAD>
<BODY>
<ELS_RSETTINGS>
    SET REPORT_TITLE      = "Simple XML Report";
    SET PAGE_ORIENTATION  = ELS_PORTRAIT;
    SET PAGE_SIZE         = ELS_LETTER;
    SET PAGE_SOURCE       = ELS_UPPER;
    SET PAGE_MARGINS.LEFT        = 0.5;
    SET PAGE_MARGINS.RIGHT       = 0.5;
    SET PAGE_MARGINS.TOP         = 0.75;
    SET PAGE_MARGINS.BOTTOM      = 0.5;
    SET DEFAULTMODE              = ELS_FAST;
```

```
    // define the XML document
    DECLARE @oXML XMLDOM;
    @oXML.load("C:\ELSS\Samples\Data\Orders.xml");

    // define the XML dataset
    DECLARE @i INT;
    DECLARE @oRecords XMLDOMNodeList;
    DECLARE @oRec XMLDOMNode;
    DECLARE @oFld(5) XMLDOMNode;

    SET @oRecords = @oXML.selectNodes("dataroot/Orders[(OrderID >= "
                                          + Format(OrderID,"") + ")]");

</ELS_RSETTINGS>

<ELS_RHEADER FONT-FAMILY="Times New Roman" FONT-SIZE="10pt">
<DIV align="center" height="60">
<FONT style="font-family:Arial; font-size:24pt;color:darkgoldenrod">
<B>Simple XML Report</B>
</FONT><BR><BR>
</DIV>
</ELS_RHEADER>

<ELS_PHEADER HEIGHT="30px" FONT-FAMILY="Arial" FONT-SIZE="8pt">
<TABLE style="FONT-SIZE: 8pt; WIDTH: 100%; HEIGHT: 20px">
<TBODY>
    <TR style="HEIGHT: 16px" vAlign="top">
        <TD style="FONT-WEIGHT: bold; WIDTH: 541px">
            <SPAN class="Field" style="OVERFLOW: hidden; WIDTH: 408px; COLOR: gray;
                                       WHITE-SPACE: nowrap; HEIGHT: 14px">
            <FLD>"Simple XML Report"</FLD></SPAN>
        </TD>
        <TD style="WIDTH: 173px; TEXT-ALIGN: right">
            <SPAN class="Field" style="OVERFLOW: hidden; WIDTH: 141px; COLOR: gray;
                                                    WHITE-SPACE: nowrap">
            <FLD>"P " + Format(PageNum(),"") + " / " + Format(PageCount(),"")</FLD></SPAN>
        </TD>
    </TR>
</TBODY>
</TABLE>
</ELS_PHEADER>

<ELS_PFOOTER HEIGHT="30px" FONT-FAMILY="Arial" FONT-SIZE="8pt">
<TABLE style="FONT-SIZE: 8pt; WIDTH: 100%; HEIGHT: 20px">
<TBODY >
    <TR style="HEIGHT: 16px" vAlign="top">
        <TD style="FONT-WEIGHT: bold; WIDTH: 541px">
            <SPAN class="Field" style="OVERFLOW: hidden; WIDTH: 408px; COLOR: gray;
                                       WHITE-SPACE: nowrap; HEIGHT: 14px">
            <FLD>"RUN DATE-TIME: " + Format(GetDate(),"mm/dd/yy")
                   + "(" + Format(GetDate(),"hh:nn:ss") + ")"</FLD>
            </SPAN>
        </TD>
        <TD style="WIDTH: 173px; TEXT-ALIGN: right">
            <SPAN class="Field" style="OVERFLOW: hidden; WIDTH: 141px; COLOR: gray;
                                                    WHITE-SPACE: nowrap">
            <FLD>" "</FLD>
            </SPAN>
        </TD>
    </TR>
</TBODY>
</TABLE>
</ELS_PFOOTER>

<ELS_RDETAIL FONT-FAMILY="Times New Roman" FONT-SIZE="8pt">
<ELS_ROW NAME="ELSRow1">
    <L border="0" CellSpacing="0" CellPadding="0" WIDTH="100%" HEIGHT="15">
        <C HEIGHT="15" WIDTH="10%">
            <HDR>
            <B>OrderID</B>
            </HDR>
        </C>
        <C ALIGN="center" HEIGHT="15" WIDTH="20%">
            <HDR>
```

```
            <B>Order Date</B>
            </HDR>
        </C>
        <C ALIGN="center" HEIGHT="15" WIDTH="10%">
            <HDR>
            <B>ShipVia</B>
            </HDR>
        </C>
        <C HEIGHT="15" WIDTH="30%">
            <HDR>
            <B>Ship Name</B>
            </HDR>
        </C>
        <C HEIGHT="15" WIDTH="30%">
            <HDR>
            <B>Ship City</B>
            </HDR>
        </C>
    </L>
    <L border="0" CellSpacing="0" CellPadding="0" HEIGHT="15" WIDTH="100%">
        <C HEIGHT="15" WIDTH="10%">
            <FLD>
            COALESCE(@oFld(0).firstChild.nodeValue, "")
            </FLD>
        </C>
        <C ALIGN="center" HEIGHT="15" WIDTH="20%">
            <FLD>
            Format( ToDATE(COALESCE(@oFld(1).firstChild.nodeValue, ""),
                                        "yyyy-mm-ddThh:nn:ss"),"mm-dd-yyyy" )
            </FLD>
        </C>
        <C ALIGN="center" HEIGHT="15" WIDTH="10%">
            <FLD>
            COALESCE(@oFld(2).firstChild.nodeValue, "")
            </FLD>
        </C>
        <C HEIGHT="15" WIDTH="30%">
            <FLD>
            COALESCE(@oFld(3).firstChild.nodeValue, "")
            </FLD>
        </C>
        <C HEIGHT="15" WIDTH="30%">
            <FLD>
            COALESCE(@oFld(4).firstChild.nodeValue, "")
            </FLD>
        </C>
    </L>
</ELS ROW>

<ELS>
    BeginHeader("ELSRow1");
    WHILE @i < @oRecords.length
            SET @oRec = @oRecords.item(@i);

            SET @oFld(0) = @oRec.selectSingleNode("OrderID");
            SET @oFld(1) = @oRec.selectSingleNode("OrderDate");
            SET @oFld(2) = @oRec.selectSingleNode("ShipVia");
            SET @oFld(3) = @oRec.selectSingleNode("ShipName");
            SET @oFld(4) = @oRec.selectSingleNode("ShipCity");

            ResultRow("ELSRow1");

            SET @i = @i + 1;
    END LOOP
    EndHeader("ELSRow1");
</ELS>
</ELS_RDETAIL>

</BODY>
</HTML>
```