Expanding Blue Chameleon : OSL Scripting Language Documentation

November 14, 2013

Chapter 1

The OSL Scripting Language

OSL is a scripting language originally developed for building **Blue Chameleon** ; its high versatility, as well as its ease at incorporating and handling other languages such as HTML and SQL, make it the first choice whenever you may need to develop any web-based application - no matter how complex - dealing with database processing and displaying.

Reference Web Page : `http://www.inc.lu/OSL/script.htm`

1.1 Writing and running OSL scripts

OSL scripts are written into `.phs` files. They can be typed in any text editor, preferably with HTML highlighting.

`.phs` files are run automatically whenever called by Blue Chameleon or by another `.phs` file.

1.2 Syntax overview

1.2.1 Commands

OSL is mainly built around **commands**, which consist in a `COMMAND` name (always in uppercase) followed (or not) by one or several arguments, sandwiched between opening and closing guillemots «. . . ».

For instance :

```
«SYSTEM ECHO "Hello world !"»
```

which outputs string "Hello world !" on the system.

Commands can also be made of an opening and a closing tag :

```
«IF <condition>»
  [Do stuff]
«ENDIF»
```

1.2.2 Functions

OSL also makes use of functions, which aim is to achieve various operations on variables such as string manipulations, date operations... A function name is always in lowercase, preceded by @ and followed by its argument(s) between parentheses. If function requires several arguments, they are separated by semicolons. For instance, this call :

```
«VAR NEW _Trunc=@substr("Hello world"; 6; 5)
```

which stores into variable _Trunc 6 characters from string "Hello world" starting from the 5th position.

It is to note that any result of a function call can be assessed to a variable of the right type : myDouble=@atof("12.1") stores for example 12.1 into double myDouble.

To evaluate *and* display any function result, command EXPR is used :

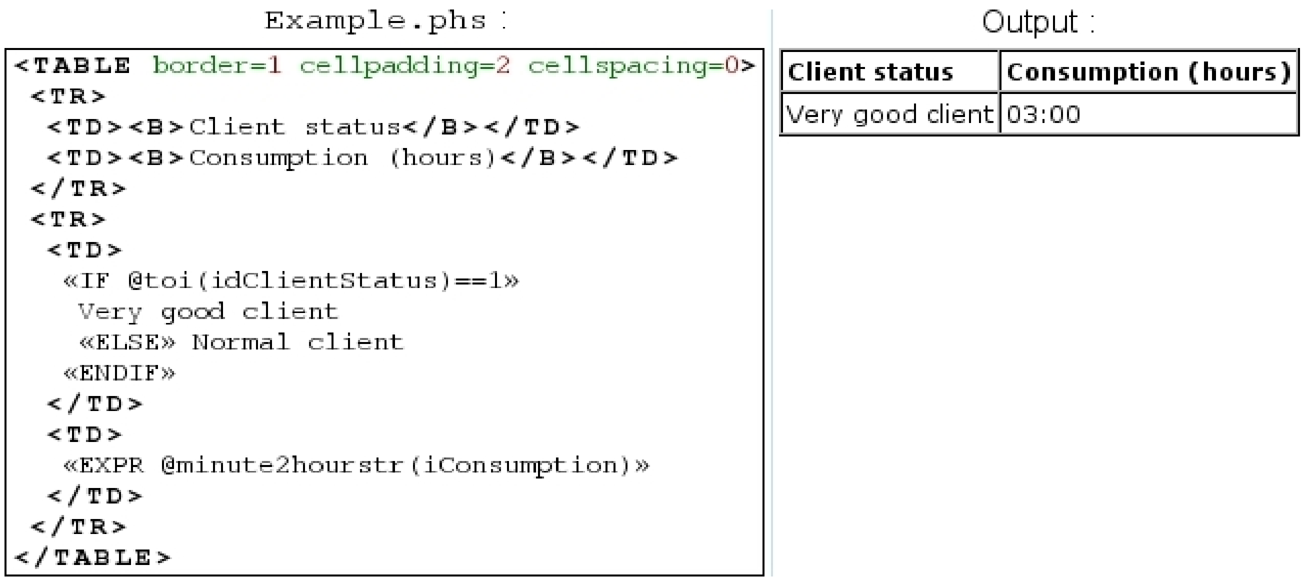
```
«EXPR @atof("12.1")»
```

which outputs 12.1. Or even simpler, for the same result :

```
«=@atof("12.1")»
```

1.2.3 HTML code

HTML code is used like in any HTML editor and it is possible to use commands and functions within it, as featured in Fig.1.1.



```
<TABLE border=1 cellpadding=2 cellspacing=0>
 <TR>
  <TD><B>Client status</B></TD>
  <TD><B>Consumption (hours)</B></TD>
 </TR>
 <TR>
  <TD>
   «IF @toi(idClientStatus)==1»
    Very good client
    «ELSE» Normal client
   «ENDIF»
  </TD>
  <TD>
   «EXPR @minute2hourstr(iConsumption)»
  </TD>
 </TR>
</TABLE>
```

Client status	**Consumption (hours)**
Very good client	03:00

Figure 1.1: HTML code and OSL commands and functions can be mixed in a very straightforward way.

1.2.4 Comments

They can be added anywhere via

```
«* Now processing default case...  *»
```

1.3 `OSL` variables

1.3.1 Managing variables

1.3.1.1 Declaring and assigning

Variables can be declared anywhere in a `.phs` file with commands `VAR NEW` for local variables and `LET` for global variables ; for both, values can be assigned right at the declaring. For instance :

```
«VAR NEW dMyDouble=200.0»
```

```
«LET _MyString="A global string..."»
```

It is to note that `LET` is also used to assign a value to a pre-existing variable, whether local or global.

1.3.1.2 Test of existence

Before using a variable in a script where it is not sure if it has been transmitted as a CGI or not, function `@exists("<variable>")` is useful to test whether this variable does exist, and if not, initialize it :

```
«IF @exists("_DisplayAll")==0»
 «VAR NEW _DisplayAll=0»
«ENDIF»
```

1.3.1.3 Local environments

Between the `«VAR LOCAL»` and `«VAR ENDLOCAL»` tags, one can define an environment for local variables, even if those already existed before. If so, at the end of this environment, such an already-existing variable will get its former value back.

1.3.1.4 Deleting variables

If needed, a variable `<var>` can be deleted through `«VAR DELETE <var>»`.

1.3.2 Types

Declaring a variable's type is not mandatory ; however, it can be done through

```
«VAR NEW [double]dMyDouble=200.0»
```

In `OSL` the following types are used :

- `int` for integers ;
- `int64` for 64-bit integers ;
- `double` for double-precision floating-points numbers ;
- `string` for strings, always declared/assigned between double quotes : "...".

In this documentation, variables are represented as `<value:type>`.

1.3.2.1 Casting

Several functions exist to cast variables :

- `@atoi(<value:string>)` converts a string to the integer value ; if unsure of the variable type - string or integer -, function `@toi(<value:string or integer>)` might rather be used. This latter function must be for instance **called before using a integer CGI variable**, as the previous URL redirection (`...&_intCGIVar=25&...`) has cast it into a string :

  ```
  «LET _intCGIVar=@toi(_intCGIVar)»
  ```

- `@atof(<value:string>)` converts a string to the floating-point value ;
- `@itoa(<value:int>)` converts an integer to a string ; if unsure of the variable type - integer or string -, function `@toa(<value:integer or string>)` might rather be used.

This example

```
«VAR NEW _fiveString="5"»
«VAR NEW _fiveInt=5»
Converting _fiveString to integer :  «=@toi(_fiveString)+1»
Converting _fiveInt to string :  «=@toa(_fiveInt)+"+1"»
```

will then output

```
Converting _fiveString to integer : 6
Converting _fiveInt to string : 5+1
```

1.3.3 Naming

While the naming of `OSL` variables is free, it is advised to follow some simple guidelines in order to distinguish at a glance what is the type of a variable :

- integers can be named with a 'i' or 'id' prefix (e.g. `iZipCode`, `idClient`) ;
- doubles can be named with a 'd' prefix (e.g. `dOrderAmount`) ;
- strings can be named with a 'az' prefix (e.g. `azClientFirstName`).

Most usually, an underscore is put before a variable's name when `SQL` queries are involved (1.6) ; e.g., so that variable `_idClient` is distinguished from field name `idClient`.

1.3.4 Evaluating variables

There are two ways of evaluating a variable `<var>` : either by command `EXPR` or by placing it between guillemots : `«<var>»`.

When `<var>` is already in a `«...»` context (for instance in `«IF <var>==1»`), there is no need to use those ; otherwise, anytime when <var> is outside those and needs to be evaluated, guillemots are used, for instance :

```
<TABLE border=«MyOwnBorder» cellpadding=2 cellspacing=0>
```

1.3.5 Predefined variables

There exists various system variables that are called to give information about statuses, success or failure of operations... Most of them are dedicated to `OSL`'s `SQL` functions (1.6).

All the names of the predefined variables are in uppercase, sandwiched between two #. They cannot be assigned any value. For example,

```
«EXPR #SQLSTATUS#»
```

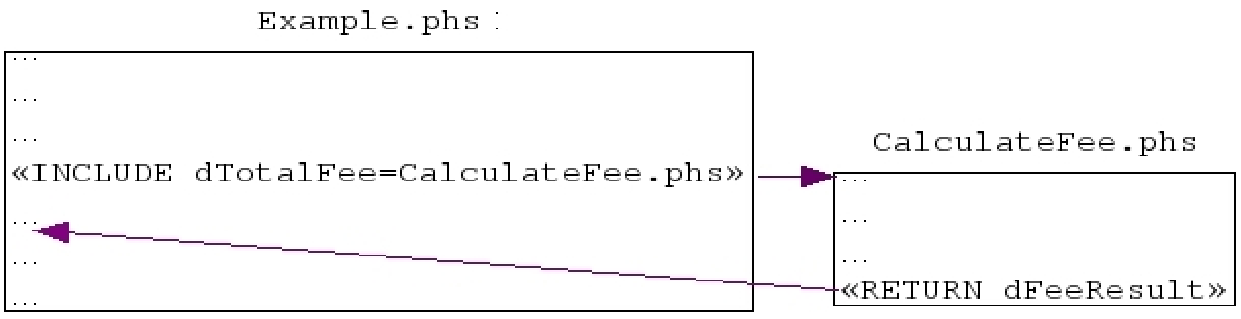
or

```
«VAR NEW _Succ=#SQLSTATUS#»
```

outputs/stores whether the last `SQL` operation has succeeded (returning 1) or failed (returning 0).

1.4 Including : script files, procedures

As the length of a `.phs` file can grow rapidly, it is clever to split tasks into smaller files and procedures. Those can then be called anywhere, with certain limits.



1.4.1 Calling other files

The «INCLUDE <scriptfile>» command simply calls the contents of file <scriptfile>. If this file happens to RETURN a value, it can be assessed to a variable of the calling file by precising its name in the command :

It is to note that whenever RETURN is called, the script is exited.

1.4.1.1 Passing variables to a called script

It is possible to pass a list of variables to be used in the called script, for instance :

```
«INCLUDE dMonthlyFee=MonthlyFee.phs;_azMonth="February";_azStatus="VIP"»
```

There, the MonthlyFee.phs script has two string variables _azMonth and _azStatus that are initialized to fixed values "February" and "VIP" and used to perform calculations accordingly, thus returning a result dMonthlyFee.

If the values to be given are variables themselves, they are passed with guillemots, **unless it is a string** ; for instance :

```
«INCLUDE modifyName.phs;_ClientId=«_idClient»;_NewName=_ThisName»
```

In this example, integer value _idClient and string value _ThisName serve as to assess values to variables _idClient and _NewName in script modifyName.phs. Operations there (for instance a SQL table update) will be performed using those two imputed values.

1.4.1.2 Persistence of variables as used in a called script

It is to note that, if some variables as used in a called script are initialized before the call, they are kept after the call. It is for instance used when getting user-related information (B.6.1) :

```
«VAR NEW _UserRealName=""»
«VAR NEW _UserFirstName=""»

«INCLUDE
ossbo:OSSUserInfo.phs;_MerchantId=«_MerchantId»;_UserId=«#SQLUSERID#»»

Current user :  «_UserFirstName»  «_UserRealName»
```

1.4.2 Calling procedures

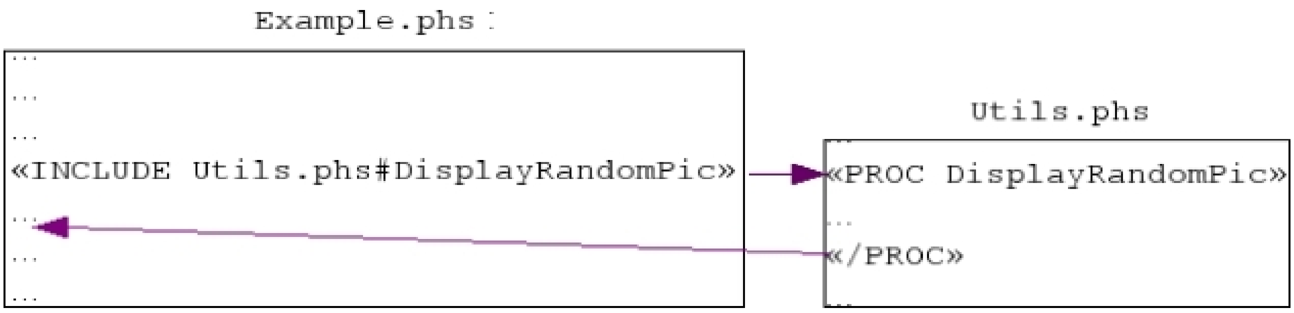
A procedure, in the `OSL` context, is a list of instructions placed between the `PROC <procedure_name>` and `/PROC` tags. It is then simply executed by command line

```
«INCLUDE <procedure_name>»
```

.

1.4.2.1 Calling procedures from other files

The example above supposes that the called procedure is defined in the same file as the calling script. It is also possible to call a procedure belonging to another script by precising its name (preceeded by #) right after called script's name :



```
...
...
...
«INCLUDE Utils.phs#DisplayRandomPic»
...
...
...
```

```
...
«PROC DisplayRandomPic»
...
«/PROC»
...
```

1.4.3 Including other files

What preceded dealt with the including of `.phs` files ; it can be said that other files can be included can be included, in particular `HTML` files, with the `INCLUDE` command, for example :

```
«INCLUDEFILE HeaderLayout.html»
```

Command `LIBRARY` fills the same role.

Refer to *The structure of a Blue Chameleon Add-on* (B.1) to know where to upload your external files.

1.5 Programming instructions

`OSL` handles usual programming instructions such as variable tests and loops.

1.5.1 Tests

Command `«IF <condition>»`, eventually terminated by an `ENDIF`, is used to perform instructions according the veracity of `<condition>`. It can be enriched with an `ELSE` followed by other instructions to perform if `<condition>` was false, as well as an `ELSEIF` stating a new condition to test for :

```
«IF <condition>»
  <instructions>
«ELSE»
  <other instructions>
«ENDIF»

«IF <condition1>»
  <instructions1>
«ELSEIF <condition2>»
  <instructions2>
«ELSEIF <condition3>»
  <instructions3>
«ENDIF»
```

An `ELSEIF` command may be followed by an `ELSE` one, but not the other way around (which would be equivalent to two consecutive *else*s).

The following table sums up the `OSL` syntax of numerical and logical tests :

Test	`OSL` **syntax**
A equal to B	`A==B`
A different from B	`A<>B`
A greater than B	`A>B`
A less than B	`A<B`
A greater than or equal to B	`A >= B`
A lesser than or equal to B	`A <= B`
A AND B	`A && B`
A OR B	`A || B`

1.5.2 Loops

A while-type loop can be implemented in the following way :

```
«WHILE»
 <instructions>
«/WHILE <condition>»
```

Instructions between the two tags are executed as long as condition is true. The following example, where instructions are repeated N times, then shows how a for-loop can be implemented :

```
«LET _Counter=0»
«WHILE»
  <instructions that may modify _Counter's value...>
  «LET _Counter=_Counter+1»
«/WHILE _Counter<N»
```

Another way of performing a loop with defined repetitions is using the `SQLREPEAT` command, for example :

```
«SQLREPEAT»
  <instructions>
«/SQLREPEAT <begin> ; <end>»
```

will perform `instructions` end-begin+1 times.

The `BREAK` command can be used to escape the `WHILE` and `SQLREPEAT` loops. It is useful for instance in long loops where just one test is needed to have confirmation that some test is fulfilled :

```
«VAR NEW _Found=0»
«SQLREPEAT»
  <some instructions to fetch a test value _TestValue, for instance
   the result of a SQL query using #SQLREPEAT# value>
  «IF _TestValue==_AimedValue»
   «LET _Found=#SQLREPEAT#»
   «BREAK»
  «ENDIF»
«/SQLREPEAT 1;10000»
```

There, the `SQLREPEAT` loop will stop some *as soon as the `_AimedValue`* is found, also keeping where it found it (the `#SQLREPEAT#` value as stored in `_Found`, now not null anymore).

1.5.2.1 Loop counter

The predefined variable `#SQLREPEAT#` can be used to access the value of the loop counter which :

- in case of a `WHILE` loop, is initialized at 1 at the beginning of the loop and is incremented by one at each repetition ;
- in case of a `SQLREPEAT` loop, is initialized at `<begin>` at the beginning of the loop and is incremented by one until reaching `<end>`.

The loop counter predefined variable may be useful to implement a for-loop, using condition `FOREVER` (which creates a dead-end loop) along with the `BREAK` command :

```
«SQLREPEAT»
  <instructions>
  «IF #SQLREPEAT#==10»
   «BREAK»
  «ENDIF»
«/SQLREPEAT FOREVER»
```

1.6 OSL and SQL

One of OSL's greatest features lies in its straightforward way to handle databases and tables ; indeed, varied functions dedicated to SQL operating fulfill all needs associated with database handling.

1.6.1 Performing a SQL command

1.6.1.1 Result-less queries

A SQL query without expected result (such as an insert) can be simply done via OSL's SQL command, for instance :

```
«SQL insert into COUNTRIES values (15,'Finland')»
```

The success or failure of lastly done SQL command can be accessed by evaluating predefined variable #SQLSTATUS#, equal either to 1 or 0.

1.6.1.2 Queries with results

According to the expected result of the SQL query to execute - integer, double or string -, three commands SQLEXEC INT, SQLEXEC DOUBLE or SQLEXEC STRING are available. For example,

```
«SQLEXEC DOUBLE dPrice=select dItemPrice from ITEMS where idItem=2»
The price of the Arrow is $ «dPrice».
```

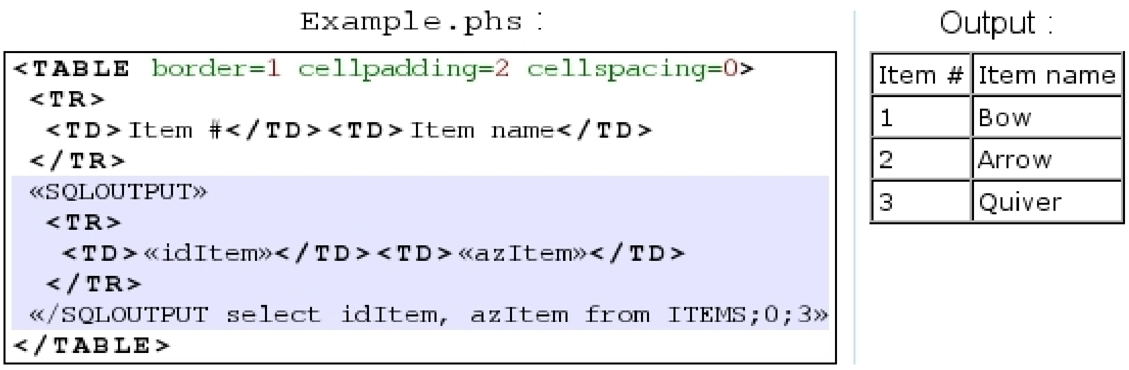
outputs The price of the Arrow is $ 3..

1.6.2 Displaying contents of a SQL table

Showing the contents of a table is a very usual operation that OSL performs through command tags «SQLOUTPUT» and «/SQLOUTPUT <SQL select query>». In between those tags are placed the names of the columns to display (always between guillemots) and the SQL can be followed of two optional numbers.

For instance, Fig.1.2 shows how two columns of a table are displayed ; the '0' after the SQL query specifies that no result records have to be skipped while the '3' specifies that three (at maximum) records have to be given.
In order for all records to be output, /SQLOUTPUT select idItem, azItem from ITEMS (i.e., without anything after the query) is used.

The «SQLOUTPUT» and «/SQLOUTPUT <SQL select query>» constitute a loop than can be conditionally exited thanks to the «BREAK» line :


Example.phs :

```
<TABLE border=1 cellpadding=2 cellspacing=0>
 <TR>
  <TD>Item #</TD><TD>Item name</TD>
 </TR>
 «SQLOUTPUT»
  <TR>
   <TD>«idItem»</TD><TD>«azItem»</TD>
  </TR>
 «/SQLOUTPUT select idItem, azItem from ITEMS;0;3»
</TABLE>
```

Output :

Item #	Item name
1	Bow
2	Arrow
3	Quiver

Figure 1.2: Dropping the three first rows of the idItem and azItem columns of the ITEMS table.

```
«VAR NEW _Found=""»
«SQLOUTPUT»
  «IF dItemPrice>100.0»
   «LET _Found=azItem»
   «BREAK»
  «ENDIF»
«/SQLOUTPUT select azItem, dItemPrice from ITEMS order by dItemPrice
   asc»

«IF _Found<>""»
  The cheapest item above 100.0 is «_Found»
«ELSE»
  No item above 100.0 !
«ENDIF»
```

In this example, records consisting in item names and prices are processed with ascending `dItemPrice` values ; as soon as a price is over 100.0 (if ever), item name is stored in variable `_Found` (now not an empty string anymore) and `SQLOUTPUT` loop, now pointless, is exited.

1.6.3 SQL transactions

`OSL` of course handles the concept of SQL transactions with the `«SQLSTATUS TRANSACTION BEGIN»` and `«SQLSTATUS TRANSACTION END»` commands, between which `«SQL ...»` commands are placed. For instance,

```
«SQLSTATUS TRANSACTION BEGIN»
  «SQL delete from ITEMS where idItem=«idOut»»
  «SQL insert into OUTDATED_ITEMS values («idOut»,«azOut»,«dOut»)»
  «VAR NEW _TransacSuccess=#TRANSACTIONSTATUS#»
«SQLSTATUS TRANSACTION END»
```

ensures that the two `SQL` commands in the middle are both successful before being committed ; otherwise they will be rolled back. In the former case, the `#TRANSACTIONSTATUS#`

predefined variable as estimated just before the end is equal to 1, and to 0 in the latter.

A rollback can also be forced by command line `«SQLSTATUS ROLLBACK»`.

1.6.3.1 Disabling autocommitting

With command line `«SQLSTATUS AUTOCOMMIT OFF»`, it is possible to disable the committing unless system encounters a `SQLSTATUS TRANSACTION END` ; otherwise, a rollback will be done.

1.6.4 Useful things to know

`SQL` queries need most of the times some of the current script variables to be used ; for instance,

```
«SQL insert into COUNTRIES values («_CountryID»,'«_CountryName»')»
```

is the way of inserting into the `COUNTRIES` table the numerical value as stored in `_CountryID` and the string as stored in `_CountryName`. In the case of the string, notice the guillemots within the single quotes.

1.7 Use of files

During the execution of a script, a non-script file may have to be opened or written to.

1.7.1 Creating and/or opening a file

Command `FILE CREATE` allows to create a file, for instance

```
«FILE CREATE Dump.txt»
```

Once created, this file is opened by

```
«FILE OPEN <mode> <filename>»
```

where `<mode>` is chosen amongst these following options :

- `READ` : for a read-only file ;
- `READTEXT` : for a read-only text file ;
- `WRITE` : for a file that is to be read and written to ;
- `WRITETEXT` : for a text file that is to be read and written to ;
- `APPEND` : for a file that is to be read and appended to ;
- `APPENDTEXT` : for a text file that is to be read and appended to.

Also, if file did not exist, command `FILE OPEN` creates it.

1.7.2 Reading from a file

Once opened, a file does not need to be called by name to be read from.

There exist different ways of reading a file :

- command line «`FILE READ 20 Container`», for instance, reads the next 20 characters from the currently opened file and stores them in variable `Container` ;
- command line «`FILE READLINE Container`», on the other hand reads and stores into `Container` the current line of the file ;
- command line «`FILE READFILE Container`» reads until the end of the file, storing it in variable `Container`.

The success of these three operations is stored into predefined variable `#SQLSTATUS#`.

1.7.3 Writing into a file

Similarly as for reading, once the file is opened, its name does not need to be mentioned for the `<data>` to be written to it.

The following writing operations are possible :

- command line «`FILE WRITE <data>`» writes the value of `<data>` to the currently opened file ;
- command «`FILE WRITETEXT "..."`» does the same, but using text only, to be put in quotes ;
- command «`FILE WRITESTRING <data>`» can also be used to write `<data>` as a string ;
- command «`FILE WRITELINE <data>`» is similar to `FILE WRITE` but also adds a line end ;
- a line end alone, without added data, can be also achieved with «`FILE WRITENEWLINE`».

The success of these writing operations is stored into predefined variable `#SQLSTATUS#`.

1.7.4 Closing a file

The command line «`FILE CLOSE`» is enough to close the currently opened file.

1.8 Various mathematical functions

Table 1.1 lists the functions that `OSL` provides for basic number operation.

Table 1.1: Various mathematical functions

`@abs(<val:integer>)`
returns the absolute value of `val` `«=@abs(-10)»` → 10
`@int(<val:double>)`
returns the integer value of `val` `«=@int(-3.14)»` → 3
`@random(<val:double>)`
returns a random integer number between 0 and `val` `«=@random(1000)»` → 698
`@round(<number:double>; <decimals:integer>)`
returns `number` as rounded to `decimals` position(s) `«=@round(3.14159;2)»` → 3.14

1.9 Date and Time functions

`OSL` handles date and time in these following formats :

- expressed as values of year, month, day, hour, minute and second ;
- using 1/1/1970 (Unix time) as a reference ;
- in Excel "Serial" format.

1.9.1 Getting current date/time

Obtaining current date/time can be performed in various outputs with three argument-less functions, as shown in Table 1.2.

Table 1.2: Functions returning the current date/time in various formats :

Functions	**Output**
`@today()`	Serial
`@daytime()`	Number of days elapsed since 1-1-1970
`@time()`	c-time

Examples :

Code	**Output**
`«LET Today=@today()»`	
`«Today»`	40084.38
`«LET cTime=@time()»`	
`«cTime»`	1254121956

1.9.2 Getting any date/time

As featured in Table 1.3, functions `@date` and `@timetime` return any date/time given in days, months, years,... into Serial and c-time date/time.

Table 1.3: Functions returning a given date/time in various formats :

Functions	**Output**
`@date(<day:integer>; <month:integer>; <year:integer>)`	Serial
`@timetime(<day:integer>; <month:integer>; <year:integer>; <hour:integer>; <minute:integer>; <second:integer>;)`	c-time

Examples :

```
Code                                           Output
«LET Date=@date(22;09;2003)»
«Date»                                         37886.00

«LET cTime=@timetime(22;09;2003;17;32;00)»
«cTime»                                        1064244720
```

Table 1.4: Date- and Time- converting functions using 1-1-1970 as a reference, either in days elapsed since or seconds (c-time) :

Functions using days elapsed since 1-1-1970	**Output**
`@daytimeyear`	Year (1970 -)
`@daytimemonth`	Month (1 - 12)
`@daytimeday`	Day (1 - 31)
`@daytimetotime`	c-time

Functions using a c-time	**Output**
`@timestr(<ctime:integer>; <format:integer>)`	see table 1.5
`@timeserial`	Serial
`@timeyear`	Year (1970 -)
`@timemonth`	Month (1 - 12)
`@timeday`	Day (1 - 31)
`@timehour`	Hour (0 - 23)
`@timeminute`	Minute (0 - 59)
`@timesecond`	Second (0 - 59)

Examples :

Code	**Output**
`«LET DateStr=@timestr(1064244720;6)»`	
`«DateStr»`	22/09/2003
`«LET Mnt=@timeminute(1064244720)»`	
`«Mnt»`	32

1.9.3 Functions using Unix time reference

Table 1.4 features extensively the `OSL` date and time functions that use the Unix time reference, with argument either in days elapsed since 1-1-1970 or seconds (c-time).

Table 1.5: `format` options for function `@timestr(<ctime:integer>; <format:integer>)` :

`format`	**Output**
1	DMY (e.g. 27/9/2009)
2	MDY
3	YMD
4	DMYHM
5	RFC822
6	DDMMYYYY (e.g. 27/09/2009)
7	MMDDYYYY
8	YYYYMMDD

1.9.4 Functions using Serial date/time

Table 1.6 features extensively the `OSL` functions that use a serial time, to output various date/time components (corresponding year, month, week,...)

Table 1.6: Date- and Time- converting functions using a serial date-time (double) :

Functions	**Output**
`@sqlstrdate`	`SQL` format
`@year`	Year (1899 -)
`@month`	Month (1 - 12)
`@week`	Week (1 - 52)
`@weekday`	Weekday (0 - 6 ; 0 : Monday, 1 : Tuesday...)
`@day`	Day (1 - 31)
`@hour`	Hour (0 - 23)
`@minute`	Minute (0 - 59)
`@second`	Second (0 - 59)

Examples :

Code	**Output**
`«LET Today=@today()»`	
`«LET Day=@day(Today)»`	
`«LET Month=@month(Today)»`	
`«LET Year=@year(Today)»`	
`«Day» «Month» «Year»`	28 9 2009

1.10 String functions

`OSL` provides various string-related functions.

1.10.1 Declaring, concatenating

A empty string is declared the following way :

```
«VAR NEW _NewString="" »
```

Two (or more) existing strings are simply concatenated via operator '+' :

```
«VAR NEW _String1="Dog"»
«VAR NEW _String2="Cat"»
«VAR NEW _String3=_String1+" & "+_String2»
```

1.10.2 Tests on a single string

Functions `@strlen` and `@type`, operating on a single string, respectively return the length of the string and its type of variable (returning 0 if variable does not exist, 1 for an integer, 2 for a double, 3 for a string and 4 for a date). For instance:

```
«=@strlen("Hello world !")» → 13
«=@type("B52")» → 3
```

1.10.3 Extracting parts of a string

1.10.3.1 To and from any position

Function `@substr(string; <offset:integer>; <count:integer>)` reads `count` characters into `string` from position `offset` ; `count` is set to -1 to specify end of string. For instance :

```
«=@substr("Hello world !";0;4)» → Hell
```

Table 1.7: String extracting functions

`@strthis(string; <char:integer>)`	
	returns a copy of `string` where the first occurrence of `char` is replaced by a string end : `«=@strthis("Hello World !";111)»` → Hell
`@strrthis(string; <char:integer>)`	
	returns a copy of `string` where the last occurrence of `char` is replaced by a string end : `«=@strrthis("Hello World !";111)»` → Hello W
`@strnext(string; <char:integer>)`	
	returns the part of `string` after the first occurrence of `char` : `«=@strnext("No easy way out";32)»` → easy way out
`@strrnext(string; <char:integer>)`	
	returns the part of `string` after the last occurrence of `char` : `«=@strrnext("No easy way out";32)»` → out

1.10.3.2 To and from any character

The four functions `@strthis`, `@strrthis`, `@strnext` and `@strrnext` all extract a part of a string based on the search of a character, given as its `ASCII` decimal value. Table 1.7 sums up how they work, with 111 and 32 being the ASCII codes for letter 'o' and 'Space'.

1.10.4 Comparison of two strings

1.10.4.1 Equality

Testing for equality of two strings `_String1` and `_String2` is simply done via '==' :

```
«VAR NEW _Equal=0»
«IF _String1==_String2»
 «LET _Equal=1»
«ENDIF»
```

1.10.4.2 Containment

`@strstrsearch(string; search_string)` looks for the presence of `search_string` in `string`, returning :

- if found, the position of the first occurrence of `search_string` in `string` (0 if right at the beginning of `string`) ;
- -1 if not found.

For instance :

`«=@strstrsearch("Hello world !";"Hell")»` → 0

1.10.5 String manipulations functions

These functions act on a string's contents, in various ways : replacing its contents, changing it to lower or uppercase...

1.10.5.1 Shifting to lower-/uppercase

The shifting of a string from lower- to uppercase and vice-versa is performed by function `@toupper` (viz. `@tolower`). For instance :

`«=@toupper("2000 Light-Years From Home")»` → 2000 LIGHT-YEARS FROM HOME

`«=@tolower("2000 Light-Years From Home")»` → 2000 light-years from home

These functions only act on alphabetic characters.

1.10.5.2 Simplifying strings

The aim of the following functions is to remove spaces, separators,... to make string comparison easier.

- `@trim` removes spaces at the beginning and end of a string and, inside the string, removes all consecutive spaces to let just one :

  `«=@trim(" too    much   spacing   ")»` → too much spacing

- `@strtoname` removes all dots from a string so that it may be used as a variable name :

  `«=@strtoname("A.New.Var")»` → ANewVar

- `@tocompare` shifts all of the string's alphabetic characters to uppercase, removing accents and eliminating separators such as spaces, dot, commas...

  `«=@tocompare("6 O’clock, Jo’s Café")»` → 6OCLOCKJOSCAFE

- `@tosqlcompare` is similar to `@tocompare`, except that the underscore and percentage character are conserved :

  `«=@tosqlcompare("...a 20% bargain")»` → A20%BARGAIN

1.10.5.3 Replacing parts of a string

Two functions fulfill replacing tasks for a string, whether of a character by another, or of a substring by another

- `@strreplace(string; <old char:integer>; <new char:integer>)` replaces every occurrence of `old char` by `new char` (both of them given as `ASCII`) ; for instance, with 100 and 103 being ASCII codes for letters 'd' and 'g' :

  `«=@streplace("Blue drapes";100;103)»` → Blue grapes

- `@strstrreplace(string; <old char:integer>; <new char:integer>)` replaces every occurrence of `old string` by `new string` ; for instance :

`«=@strstrreplace("Inflation increase";"in";"de")»` → Inflation decrease

1.10.6 Use of strings with special characters

1.10.6.1 In `SQL` queries

If the string to be inserted in a `SQL` contains special characters (such as the single quote - which would result in the query failing), the function `@atosql()` should be used. The following query is ensured to never fail whatever the value of `_CountryName` is :

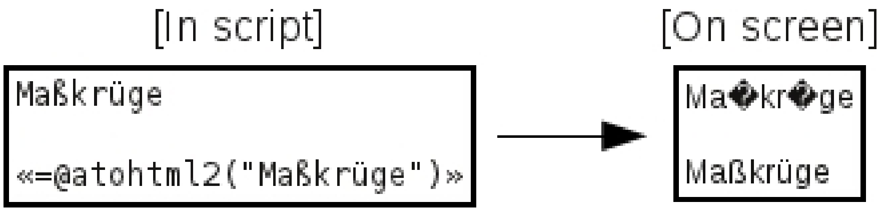
```
«SQL insert into COUNTRIES values («_CountryID»,'«=@atosql(_CountryName)»')»
```

1.10.6.2 For display on the screen

When a string variable belongs to a script called by an Ajax request, display problems might surface if string contains non-ASCII characters such as letters with diacritics.

To circumvent this, functions `@atohtml` and `@atohtml2` can be used : the former indeed converts certain caracters (<, >, &) of the string to `HTML` character sequences (`&lt;`, `&gt;`, `&amp;`) while the latter transforms all characters into their `HTML` equivalent, to be then interpreted by the browser.

In the following example, text as shown through an Ajax request has display issues which are corrected thanks to `@atohtml2` :



```
Maßkrüge

«=@atohtml2("Maßkrüge")»
```

```
Ma�kr�ge

Maßkrüge
```

This function can be also used in the shorter form `«%html;=_MyString»` (B.3.3).

1.10.6.3 For `Javascript`

Should an `OSL` string be used in a `Javascript` environment or as an argument for a `JS` function call, premature string termination problems will arise if string contains characters ' and/or `"`. To be avoid this, function `@atojavascript` is to be used as it indeed prefixes single and double quotes (also backslash) as present in the string with a '\'.

For instance, `«=@atojavascript("Patrick O'Hara")»` would output `Patrick O\'Hara'`.

This function can be also used in the shorter form `«%js;=_MyString»` (B.3.3).

1.10.6.4 For URLs

When a string variable is used in a URL (as a CGI variable), it might contain non-functional URL characters such as spaces and ampersands :

```
(...)
«VAR NEW _String="Smith & Wesson"»
«LET _URL=_URL+_String»
```

To make such strings work properly as URLs, the `@atocmd` function should be used : it transforms spaces by the character '+' and +, &, " and all caracters above decimal code 127 by their %hh hexadecimal codes. In the example above, doing `«VAR NEW _String=@atocmd(_String)»` (after its initializing) will indeed assess to `_String` the new value "Smith+%26+Wesson". Function `@atocmd2` is similar, except that it transforms spaces into %20.

1.10.6.5 Various string functions

Function `@strreverse` fully reverses a string :

`«=@strreverse("Not a palindrome")»` → emordnilap a toN

1.11 `MAIL` functions

These functions are used by Blue Chameleon when an email is automatically sent, for instance when an order has been done. Their simplicity makes them fully integrable in a developed script, as shown at *Using Blue Chameleon's mail gate* (B.9).