List & Label® 21

Scripting Support Documentation

Programmer's Reference

Contents

1 Introduction

Scripting in List & Label offers you a powerful expansion option that allows access to variables, fields and more. Using scripts, you can reference them in order to comfortably realize a number of additional functions in a language of your choice.

> Note: A script is a sequence of commands which are processed sequentially upon execution. The commands are drawn from the "vocabulary" of a particular script language. This command set determines which possibilities the language offers and how a script has to be constructed.

Scripts are generally not too complex and can produce remarkable results with just a few commands. An average script comprises around 20 to 40 lines of commands. For these reasons, among others, script languages are usually very easy to learn.

Although they look very similar on the surface, there are a considerable number of crucial differences between scripts and executable programs.

For example, scripts are not executable on their own but always require an environment in which to run. *Hosts*, as such environments are called, are responsible for the administration of scripts and increase the range of possibilities for the respective language, mostly in form of additional objects. In this case, List & Label is the host. The provided framework offers the user a powerful interface for expanding the functionality of the formula editor. As List & Label scripts are commonly run within the printing loop, it should be clear that they may not contain any GUI elements. For the same reason, they should be executable within a reasonable time frame.

1.1 Which script languages are supported?

Generally speaking, all Windows Scripting Host script languages are supported in theory. The most commonly used, however, are VBScript and JScript, which are provided directly by the manufacturer, Microsoft. But there are also other implementations, e.g. Python. A parameter string is used to select the designated language when calling the script functions.

> Note: VBScript and JScript are usually pre-installed on your system. If not, or if you wish to use other languages, you must obtain them from the respective manufacturers and install them onto the system according to their specifications.

Apart from the classical Windows Scripting Host script languages, C# is also supported as a script language.

The full range of functions of the .NET Framework 4.0 is available for C# scripts, which is why this .NET Framework version or higher is the minimum requirement in order to run C# scripts.

As C# scripts have to be compiled before use and this can take longer than running the script itself in some instances, the 30 most recently compiled scripts are cached for repeat execution. The corresponding files are located in the folder "%temp%\combitCSharpScriptCacheLL21\". It contains both the file "combitCSharpScriptCache.cache", which contains information about the stored scripts, and a folder for each script named according to the following template: "combitCSharpScript_[GUID]" (e.g. "combitCSharpScript_e9527e037aa149f3ba79bd408a8232db"). This folder contains all the .dll files used by the script, debug information, and the script itself (also as a .dll file).

We also recommend testing the scripts on the target system when using Windows XP due to various different system environment states (Windows Updates, Service Packs).

1.2 How and where can scripts be integrated?

You can easily integrate scripts using the formula editor integrated into List & Label. The actual script code can either be a directly embedded part of the formula or, alternatively, external code can be referenced via the designer function LoadFile$. You can also integrate additional external code via include statements. Especially in the case of larger scripts, using external text files is a better choice, because that way they can easily be reused elsewhere if required.

The following designer functions are provided:

Script$: Returns the result of a script as a character string

ScriptBool: Returns the result of a script as Boolean

ScriptDate: Returns the result of a script as a date

ScriptVal: Returns the result of a script as a numeric value

1.3 Support for scripting functionality

The potential uses of scripting technology are so numerous that describing them would certainly warrant an entire book of its own. We hope you understand that we cannot provide a description of the script languages used. You can find relevant literature in the book catalogs of many major specialist publishers.

Microsoft provides extensive online documentation for VBScript and JScript: https://technet.microsoft.com/en-gb/scriptcenter/.

Of course, as part of our support services, we would be pleased to assist you with any questions and wishes you may have in order to provide you with an optimal experience with our product. However, we hope you understand that we can only respond to questions regarding the object model itself, but not regarding models of other products or the script languages themselves.

2 Preprocessor and options

2.1 Activating the script engine

The script engine is disabled by default for security reasons, as it allows the user to access system functions via the script language in the context of the current application user. Therefore the script engine needs to be activated first. Three options are available for this purpose.

Activate general scripting (default false)

LlSetOption(hJob, LL_OPTION_SCRIPTENGINE_ENABLED, true);

Optionally, you can set a timeout for the maximum runtime for a script (default 10000ms). A script with a longer runtime will be aborted by the environment. Be careful not to set too low a timeout threshold for C# scripting, as any required compiling time counts towards the runtime.

LlSetOption(hJob, LL_OPTION_SCRIPTENGINE_TIMEOUTMS, 15000);

Optionally, you can set the formula editor to run the expression in real time after every keystroke. However, this could place a considerable load on the system and could also be problematic depending on the script language and content, therefore the default setting is also false.

LlSetOption(hJob, LL_OPTION_SCRIPTENGINE_AUTOEXECUTE, true);

2.2 In general for all languages

All statements for the preprocessor such as pragmas, options or includes always need to be placed in a separate line in order for them to be handled correctly. Leading spaces or tabs are ignored. However, it is possible to comment them out ad hoc.

The comment sign for all preprocessor statements is // (analogous to C/C++/C#). The preprocessor does not support multi-line comment blocks.

As formula parameters within a List & Label expression are marked by either ' or ", their use within a script is limited if the source code is directly embedded into the formula. In that case you are limited to using previously unused characters within the source code. Of course, if the source code is loaded from an external file via the LoadFile$ routine instead, this restriction does not apply.

2.2.1 Selecting the script language

With the command

```
<!--#language="[script language]"-->
```

you can also explicitly set the script language within the code. Use of this command is optional. If a language is specified here, the system will only check and warn you if different languages are mixed up. The actual language selection is performed using the respective script call parameters. The identifiers required for Script$ constitute possible values, e.g. "CSharpScript", "VBScript" or "JScript".

2.2.2 Integrating scripts within scripts

It is advisable to store frequently required functions centrally, such that any necessary changes affect all scripts based on them. For this purpose, integration of scripts is supported using a special statement in the following format:

```
<!--#include file="c:\scripts\include.vbs"-->
```

The statement is thereby replaced by the complete content of the specified file. Include statements and all other pragmas and options always need to be placed in a separate line in order for the preprocessor to handle them correctly.

Note: All scripts integrated in this way must use the same script language as the primary script. Mixing of multiple languages is not permitted and leads to syntax errors.

If you stored your script below the program directory, you can use the %APPDIR% variable instead of a fixed variable:

```
<!--#include file="%APPDIR%\include.vbs"-->
```

2.3 C#-specific

2.3.1 Logging [C#]

In order to activate logging for a script, the following statement needs to be included:

```
<!--#pragma forcelogging-->
```

The log output is stored in the file "%temp%\combitCSharpScript.log". Logging can be time-consuming and therefore should not be enabled by default.

2.3.2 Debug mode [C#]

If a script contains the statement

```
<!--#pragma debugmode-->
```

a debugger is triggered at the start of the script (if one is installed on your system), with which you can check the script for errors step by step, and error messages triggered by exceptions also contain more information and line numbers.

2.3.3 Adding references [C#]

With the command

```
<!--#include ref="[file path]"-->
```

you can add external references / components, e.g. the Windows Forms Library by Microsoft (System.Windows.Forms.dll), to a script.

If you enter a file name instead of a path, an attempt will be made to load the reference from the "Global Assembly Cache". If the entire path is provided, a copy of the file will be created in a temporary folder and loaded from there.

Note: This command has to follow the preprocessor directives valid for all script languages in the script.

2.3.4 Adding namespaces [C#]

With the command

```
<!--#include using="[namespace]"-->
```

you can add using statements to the script. The advantage of doing this is that namespace names do not always have to be explicitly declared.

Instead of calling "System.Collections.Generic.List<string> obj;" for every single list, the call would only read as "List<string> obj;" after adding "System.Collections.Generic".

Note: This command has to follow the preprocessor directives valid for all script languages in the script.

3 Quick reference and examples

A script is called via the function **Script$(<language>, <code>, <opt:function>)** and returns a character string as a result. The alternative forms **ScriptVal**, **ScriptBool** and **ScriptDate** operate analogously except for the return type.

The first parameter determines the designated script language. **CSharpScript**, **VBScript** and **JScript** are the main ones supported.

The second parameter contains the script code to be executed.

The third parameter defines the return result **in VBScript**. It either contains the name of the function/method to be executed or a variable name. This third parameter is ignored **for C#**. The returning of values is handled there directly via the assignment of the WScript.Result variable.

3.1 Exemplary calls

Examples for C#:

```
Script$('CSharpScript',' WScript.Result= "language: " + Re-
port.Variable("LL.CurrentLanguage"); ')
```

```
Script$('CSharpScript', LoadFile$(ProjectPath$(false) + "Script.cs"))
```

The project "Order list with scripting.srt" is included in the additional examples of the sample application as an additional reference.

Examples for VBScript:

```
Script$('VBScript',' RetVal= "language: " + Report.Variable("LL.CurrentLanguage") ',
'RetVal')
```

```
Script$('VBScript', LoadFile$(ProjectPath$(false) + "Script.vbs"), RetVal)
```

3.2 General object model

You can access variables and methods provided by the List & Label host within the script.

3.2.1 Report Object

```
Method Report.Variable(<string>)           // Access to LL variable, read-only
```

```
Method Report.Field(<string>)              // Access to LL fields, read-only
```

When accessing variables and fields, make sure to pay attention to the current context. Of course, only variables and fields actually registered in the current context are accessible.

Example:

```
var s = Report.Variable("LL.CurrentContainer");
```

```
var s = Report.Field("Orders.CustomerID");
```

```
Method Report.SetVar(<string>, <value>)     // Calls the SetVar designer function
```

```
Method Report.GetVar(<string>, <value>)     // Calls the GetVar designer function
```

Using the SetVar/GetVar designer functions, you can pass on intermediate results indirectly from one script to the next while printing. Of course, the call order (columns) is crucial here. Also please refer to the SetVar/GetVar documentation.

Example:

```
Script1 -> Report.SetVar("ResultTmp", value);

Script2 -> var StartValue= Report.GetVar("ResultTmp");
```

3.2.2 WScript Object

```
Constant WScript.FullName           // Contains the complete name of the application

Constant WScript.Name               // Contains the name of the application

Constant WScript.Path               // Contains the path for the application

Constant WScript.Version            // Contains the internal version number
```

These constants are available in the script for direct access.

Example:

```
var MyFilePath= WScript.Path + "MyFile.txt";

Variable WScript.Result
```

The WScript.Result variable is particularly important in C# scripts, because it serves as a return value for all scripts. Unlike VBScript, for example, this return value is fixed and should be assigned at least once within a script. The last assignment defines the result.

This variable is not defined for other script languages.