

---

# opsi-winst Manual (4.11.2)



uib gmbh  
Bonifaziusplatz 1b  
55118 Mainz  
Tel.: +49 6131 275610  
[www.uib.de](http://www.uib.de)  
[info@uib.de](mailto:info@uib.de)

# Contents

<b>1</b>	<b>Copyright</b>	<b>1</b>
<b>2</b>	<b>Introduction</b>	<b>2</b>
<b>3</b>	<b>Command Line Parameters</b>	<b>3</b>
3.1	Log Paths . . . . .	4
<b>4</b>	<b>Additional Configurations</b>	<b>5</b>
4.1	Central Logging of Error Messages . . . . .	5
4.2	Skinnable opsi-winst . . . . .	6
<b>5</b>	<b>The opsi-winst Skript</b>	<b>7</b>
5.1	An Example . . . . .	7
5.2	Primary and Secondary Subprograms of a opsi-winst script . . . . .	8
5.3	String Expressions in a opsi-winst Script . . . . .	9
<b>6</b>	<b>Definition and Use of Variables and Constants in a opsi-winst Script</b>	<b>10</b>
6.1	General . . . . .	10
6.2	Global Text Constants . . . . .	11
6.2.1	Usage . . . . .	11
6.2.2	Example . . . . .	11
6.2.3	System paths . . . . .	11
6.2.3.1	Base system directories . . . . .	11
6.2.3.2	Common (AllUsers) directories . . . . .	11
6.2.3.3	Current (logged in or usercontext) user directories . . . . .	12
6.2.3.4	/AllNtUserProfiles directory constants . . . . .	12
6.2.4	opsi-winst Paths . . . . .	12
6.2.5	Network Information . . . . .	13
6.2.6	Data for and from opsi service . . . . .	13
6.3	String (or Text) Variables . . . . .	14
6.3.1	Declaration . . . . .	14
6.3.2	Value Assignment . . . . .	14
6.3.3	Use of variables in String expressions . . . . .	15
6.3.4	Secondary vs. primary sections . . . . .	15
6.4	Stringlist Variables . . . . .	16

<b>7</b>	<b>Syntax and Meaning of Primary Sections of a opsi-winst Script</b>	<b>17</b>
7.1	Primary Sections	17
7.2	Parametrizing opsi-winst	18
7.2.1	Example	18
7.2.2	Specification of Logging Level	18
7.2.3	Required opsi-winst Version	19
7.2.4	Reacting on Errors	19
7.2.5	Staying On Top	20
7.3	String Expressions, String Values, and String Functions	20
7.3.1	Elementary String Values	20
7.3.2	Strings in Strings (Nested String Values)	20
7.3.3	String Concatenation	21
7.3.4	String Variables	21
7.3.5	String Functions which Return the OS Type	21
7.3.6	String Functions for Retrieving Environment or Command Line Data	22
7.3.7	Reading Values from the Windows Registry and Transforming Values into Registry Format	23
7.3.8	Reading Values from ini files	23
7.3.9	Reading Product Properties	24
7.3.10	Retrieving Data from etc/hosts	24
7.3.11	String processing	25
7.3.12	Weitere String-Funktionen	26
7.3.13	(String-) Functions for Licence Management	32
7.3.14	Retrieving Error Infos from Service Calls	33
7.4	String List Functions and String List Processing	33
7.4.1	Info Maps	33
7.4.2	Producing String Lists from Strings	39
7.4.3	Loading Lines of a Text File into a String List	40
7.4.4	Simple String Values generated from String Lists	40
7.4.5	Producing String Lists from opsi-winst Sections	41
7.4.6	Transforming String Lists	42
7.4.7	Iterating through String Lists	42
7.5	Special Commands	43
7.6	Commands for logging control	43
7.7	Commands for User Information and User Interaction	44
7.8	Commands for userLoginScripts / Roaming Profile Support	45
7.9	Conditional Statements (if Statements)	45
7.9.1	General Syntax	46
7.9.2	Boolean Expressions	46
7.10	Subprogram Calls	48
7.10.1	Syntax of Procedure Calling	48
7.11	Controlling Reboot	49
7.12	Keeping Track of Failed Installations	50

<b>8</b>	<b>Secondary Sections</b>	<b>53</b>
8.1	Files Sections	53
8.1.1	Example	53
8.1.2	Modifier	53
8.1.3	Commands	54
8.2	Patches-Sections	56
8.2.1	Example	56
8.2.2	Parameter	57
8.2.3	Commands	57
8.3	PatchHosts Sections	58
8.4	IdapiConfig Sections	60
8.5	PatchTextFile Sections	60
8.5.1	Parameter	60
8.5.2	Commands	60
8.5.3	Examples	61
8.6	LinkFolder Sections	62
8.6.1	Examples	63
8.7	XMLPatch Sections	64
8.7.1	Parameter	64
8.7.2	Structure of a XML Document	64
8.7.3	Options for Selection a Set of Elements	66
8.7.4	Patch Actions	67
8.7.5	Returning Lists to the Caller	68
8.7.6	Examples	68
8.8	ProgmanGroups Sections	68
8.9	WinBatch-Sections	69
8.9.1	Call Parameter (Modifier)	69
8.9.2	Examples	70
8.10	DOSBatch/DosInAnIcon (ShellBatch/ShellInAnIcon) Sections	70
8.10.1	Parameter	70
8.10.2	Catch the output	71
8.10.3	Examples	71
8.11	Registry-Sections	71
8.11.1	Examples	71
8.11.2	Call Parameters	72
8.11.3	Commands	72
8.11.4	Registry Sections to Patch <i>All NTUser.dat</i>	74
8.11.5	Registry Sections in Regedit Format	75
8.11.6	Registry Sections in AddReg Format	76

---

8.12	OpsiServiceCall Sections . . . . .	76
8.12.1	Call Parameters . . . . .	76
8.12.2	Section Format . . . . .	77
8.12.3	Examples . . . . .	78
8.13	ExecPython Sections . . . . .	78
8.13.1	Interweaving a Python Script with the opsi-winst Script . . . . .	79
8.13.2	Examples . . . . .	79
8.14	ExecWith Sections . . . . .	79
8.14.1	Call Syntax . . . . .	80
8.14.2	More Examples . . . . .	80
8.15	LDAPsearch Sections . . . . .	80
8.15.1	LDAP – Protocol, Service, Directory . . . . .	81
8.15.2	LDAPsearch Call Parameters . . . . .	82
8.15.3	How to Narrow the Search . . . . .	83
8.15.4	LDAPsearch Section Syntax . . . . .	84
8.15.5	Examples . . . . .	85
<b>9</b>	<b>64 Bit Support</b>	<b>86</b>
<b>10</b>	<b>Cook Book</b>	<b>90</b>
10.1	9.1. Delete a File in all Subdirectories . . . . .	90
10.2	Check if a specific service is running . . . . .	91
10.3	Script for Installations in the Context of a Local Administrator . . . . .	92
10.4	XML File Patching: Setting Template Path for OpenOffice.org 2 . . . . .	100
10.5	Retrieving Values From a XML File . . . . .	101
10.6	Inserting a Name Space Definition Into a XML File . . . . .	102
<b>11</b>	<b>Special Error Messages</b>	<b>103</b>

# Chapter 1

## Copyright

The Copyright of this manual holds the uib gmbh in Mainz, Germany.

This manual is published under the creative commons license  
*Attribution - ShareAlike* (by-sa).



A German description you will find here:

<http://creativecommons.org/licenses/by-sa/3.0/de/>

The German legally binding license:

<http://creativecommons.org/licenses/by-sa/3.0/de/legalcode>

The English description:

<http://creativecommons.org/licenses/by-sa/3.0/>

The English license: <http://creativecommons.org/licenses/by-sa/3.0/legalcode>

The opsi software is in most parts open source.

Not open source are only this new parts which are still under cofunding.  
see:

[http://uib.de/en/opsi\\_cofunding/index.html](http://uib.de/en/opsi_cofunding/index.html)

All the rest of the source code is published under the GPLv3:



The legally binding GPLv3 license:

<http://www.gnu.org/licenses/gpl.html>

The name *opsi* is a registered trade mark of the uib gmbh.

The opsi-logo is owned by the uib gmbh and may be used only with explicit permission.

## Chapter 2

# Introduction

The open source program *opsi-winst* serves in the context of opsi – open pc server integration (cf. [www.opsi.org](http://www.opsi.org)) – as the central function for initiating and performing the automatic software installation. It may also be used stand alone as a tool for setup programs for any piece of software.

*opsi-winst* is basically an interpreter for a specific, rather simple script language which can be used to express all relevant elements of a software installation.

A software installation that is described by a *opsi-winst* script and performed by executing the script has several advantages compared with installations that are managed by a bunch of shell commands (e. g. copy etc.):

- *opsi-winst* offers to log very thoroughly all operations of the installation process. The support team can check the log files, and can easily detect when errors occurred or other problematic circumstances are evolving.
- Copy actions can be configured with a great variety of options if existing files shall be overwritten
- Especially, it may be configured that files are copied depending on their internal version.
- There are different modi for writing to the Windows registry:
  - overwrite existing values
  - write only when no value exists
  - append a value to an existing value.
- The Windows registry can be patched for all users which exist on a work station (including the default user, who is used as prototype for further users).
- There is a sophisticated syntax for an integrated patching of XML configuration files.

## Chapter 3

# Command Line Parameters

*opsi-winst* can be started with different sets of parameters depending on context and purpose of use.

There are the following syntactical schemata:

(1) Show usage:

```
opsi-winst /?
opsi-winst /h[elp]
```

(2) Execute a script

```
opsi-winst <script file>
[/logfile <log file> ]
[/batch | /histolist <opsi-winst config file path>]
[/usercontext <[domain\]user name> ]
[/parameter parameter string]
```

(3) Execute a list of scripts (separated by semicolons) one by one:

```
opsi-winst /scriptfile <scriptfile> [;<script file>]* [ /logfile <log file> ]
[/batch | /silent ]
[/usercontext <[domain\]user name> ]
[/parameter <parameter string>]
```

4) Read the PC configuration from the opsi service and act accordingly, since *opsi-winst* 4.3

```
opsi-winst /opiservice <opiserviceurl>
[/clientid <clientname>]
[/username <username>]
[/password <password>]
[/sessionid <sessionid>]
[/logfile <logfile>]
[/allloginscripts]
[/silent]
[/parameter <parameterstring>]
```

Some explanations:

- Default name of the log file is `c:\tmp\instlog.txt`
- The parameterstring, which is marked by the option `/parameter`, is accessible for every called *opsi-winst* script (via the string function `ParamStr`).

Explanations to (2) and (3) :

- If option `/batch` is used, then *opsi-winst* shows only its "batch surface" offering no user dialogs. By option `/silent` event the batch surface is suppressed. Without using option `/batch` we get into the interactive mode where script file and log file can be chosen interactively (mainly for testing purposes).
- The `winstconfigfilepath` parameter which is designated by `/histofile` refers to a file in ini file format that holds the (in interactive mode) last used script file names. The dialogue surface presents a list box that presents these file names for choosing the next file to interpret. If `winstconfigfilepath` ends with `"\"` it is assumed to be a directory name and `WINST.INI` serves as file name.

Explanations to (4):

- The default for `clientid` is the full qualified computer name
- When called with option `/allloginscripts` *opsi-winst* can do configurations for the logged in user (particularly in a Roaming Profile context). This is a co-funding feature - you need to buy it in order to use it. See at the opsi-manual for more information about *Roaming-Profile Support*.
- By option `/silent` event the batch surface is suppressed.

The not interactive mode is implied.

### 3.1 Log Paths

By default log files are written into the directory `c:\tmp` which *opsi-winst* tries to create. If *opsi-winst* has no access to this directory it uses the user-TEMP directory.

The default log file name is `instlog.txt`. The log file name and location will be overwritten via the specific command line option.

In the case, that *opsi-winst* executes a script in `/batch` mode and with a specified (and working) usercontext, the default logging path is the `opsi/tmp` in the appdata directory of the user. This will be overwritten by an explicit given log path.

In addition, *opsi-winst* uses the logging directory for saving certain temporary files.

# Chapter 4

## Additional Configurations

### 4.1 Central Logging of Error Messages

If wanted, *opsi-winst* writes the error data to a second file on a network drive or sends them to a syslog demon.

The feature can be configured in the Windows registry:

In `HKEY_LOCAL_MACHINE`, we have in a standard installation the key `\SOFTWARE\opsi.org`. We can create a subkey `syslogd` with a variable `remoteerrorlogging`. Its value determines if and, if yes, by which method a central logging shall take place.

Furthermore, in

`HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org\syslogd`

we have to observe three up to three variables:

- If `remoteerrorlogging` has value 0, no extra central logging takes place (default).
- If `remoteerrorlogging` has value 2, the error reports are sent to syslog demon. The demon host name is read from the variable `sysloghost` (default localhost) , the syslog channel number can be set from the value of the variable `syslogfacility` (default 18, that is local2).

The following table shows the possible values for the facility:

<code>ID_SYSLOG_FACILITY_KERNEL</code>	= 0;	// kernel messages
<code>ID_SYSLOG_FACILITY_USER</code>	= 1;	// user-level messages
<code>ID_SYSLOG_FACILITY_MAIL</code>	= 2;	// mail system
<code>ID_SYSLOG_FACILITY_SYS_DAEMON</code>	= 3;	// system daemons
<code>ID_SYSLOG_FACILITY_SECURITY1</code>	= 4;	// security/authorization messages (1)
<code>ID_SYSLOG_FACILITY_INTERNAL</code>	= 5;	// messages generated internally by syslogd
<code>ID_SYSLOG_FACILITY_LPR</code>	= 6;	// line printer subsystem
<code>ID_SYSLOG_FACILITY_NNTP</code>	= 7;	// network news subsystem
<code>ID_SYSLOG_FACILITY_UUCP</code>	= 8;	// UUCP subsystem
<code>ID_SYSLOG_FACILITY_CLOCK1</code>	= 9;	// clock daemon (1)
<code>ID_SYSLOG_FACILITY_SECURITY2</code>	= 10;	// security/authorization messages (2)
<code>ID_SYSLOG_FACILITY_FTP</code>	= 11;	// FTP daemon
<code>ID_SYSLOG_FACILITY_NTP</code>	= 12;	// NTP subsystem
<code>ID_SYSLOG_FACILITY_AUDIT</code>	= 13;	// log audit
<code>ID_SYSLOG_FACILITY_ALERT</code>	= 14;	// log alert
<code>ID_SYSLOG_FACILITY_CLOCK2</code>	= 15;	// clock daemon (2)
<code>ID_SYSLOG_FACILITY_LOCAL0</code>	= 16;	// local use 0 (local0)
<code>ID_SYSLOG_FACILITY_LOCAL1</code>	= 17;	// local use 1 (local1)
<code>ID_SYSLOG_FACILITY_LOCAL2</code>	= 18;	// local use 2 (local2)
<code>ID_SYSLOG_FACILITY_LOCAL3</code>	= 19;	// local use 3 (local3)
<code>ID_SYSLOG_FACILITY_LOCAL4</code>	= 20;	// local use 4 (local4)
<code>ID_SYSLOG_FACILITY_LOCAL5</code>	= 21;	// local use 5 (local5)
<code>ID_SYSLOG_FACILITY_LOCAL6</code>	= 22;	// local use 6 (local6)
<code>ID_SYSLOG_FACILITY_LOCAL7</code>	= 23;	// local use 7 (local7)

## 4.2 Skinnable opsi-winst

Since version 3.6 *opsi-winst* has an adaptable skin. Its elements are located in a subdirectory *winstskin* of the directory of the executed *opsi-winst*. The definition file which you may edit is *skin.ini*.

## Chapter 5

# The opsi-winst Skript

On principle: *opsi-winst* is an interpreter for a specific, easy to use scripting language which is tailored for the requirements of software installations. A script should be an integrated description, and a means of control, for the installation of one piece of software.

The following section sketches the structure of a *opsi-winst* script. The purpose is to identify the book marks of a script: in which way we to have to look into it to understand its processing.

All elements shall be described more in detail in the further section. The purpose then will be to show how scripts can be modified or developed.

### 5.1 An Example

*opsi-winst* scripts are roughly derived from .INI files. They are composed of sections, which are marked by a title (the section name) which is written in brackets [].

Schematically a *opsi-winst* script looks like this one (here with a check which operating system is installed):

```
[Actions]
Message "Installation of Mozilla"
SetLogLevel=6

;which Windows-Version?
DefVar $MSVersion$

Set $MSVersion$ = GetMsVersionInfo
if ($MSVersion$ >= "6")
    sub_install_win7
else
    if ( $MSVersion$ = "5.1" )
        sub_install_winXP
    else
        stop "not a supported OS-Version"
    endif
endif

[sub_install_win7]
Files_copy_win7
WinBatch_Setup

[sub_install_winXP]
```

```
Files_copy_XP
WinBatch_SetupXP

[Files_copy_win7]
copy "%scriptpath%\files_win7\*.*" "c:\temp\installation"

[Files_copy_winxp]
copy "%scriptpath%\files_winxp\*.*" "c:\temp\installation"

[WinBatch_Setup]
c:\temp\installation\setup.exe

[WinBatch_SetupXP]
c:\temp\installation\install.exe
```

How can we read the sections of this script?

## 5.2 Primary and Secondary Subprograms of a opsi-winst script

The script as a whole serves as a program, an instruction for an installation process. Therefore each of its sections can be seen as a subprogram (or "procedure" or "method"). The script is a collection of subprograms.

The human reader as well as an interpreting software has to know at which element in this collection reading must start.

Execution of a *opsi-winst* script begins with working on the sections [Initial] and [Actions] (in this order). All other sections are called as subroutines from these two sections. This process is only recursive for Sub sections: Sub sections have the same syntax as Initial and Actions sections and may contain calls for further subroutines.

---

### Note

If a script is run as *userLoginScript* and it contains a section [ProfileActions], so the script interpretation will be started at the ProfileActions section.

---

This gives reason to make the distinction between primary and secondary subprograms:

The primary or general control sections comprise

- the optional **Initial** section (by convention the beginning of the script),
- the **Actions** section (should follow to **Initial** section), and
- **Sub** sections (0 to n subroutines called by the **Actions** section which are syntactical and logical extensions of the calling section).
- the **ProfileActions** section, which will be interpreted in different ways according to the script mode (Machine/Login).

The procedural logic of the script is determined by the sequence of calls in these sections.

The secondary or specific sections can be called from any primary section but have a different syntax. The syntax is derived from the functional requirements and library conditions and conventions for the specific purposes. Therefore no further section can be called from a secondary section.

At this moment there are the following types of secondary sections:

- Files sections,

- WinBatch sections,
- DosBatch/DosInAnIcon/ShellInAnIcon sections,
- Registry sections
- Patches sections,
- PatchHosts sections,
- PatchTextFile sections,
- XMLPatch sections,
- LinkFolder sections,
- opsiServiceCall sections,
- ExecPython sections,
- ExecWith sections,
- LDAPsearch sections.

Meaning and syntax of the different section types are treated in Chapter 7 and Chapter 8.

### 5.3 String Expressions in a opsi-winst Script

Textual values (string values) in the primary sections can be given in different ways:

- A value can be directly cited, mostly by writing in into (double) citation marks. Examples:  
*"Installation Of Mozilla"*  
*"n:\home\user name"*
- A value can be given by a String variable or a String constant, that "contains" the value:  
 The variable *\$MsVersion\$* may stand for "6.1" – if it has been assigned beforehand with this value.
- A function retrieves or calculates a value by some internal procedure. E. g. **EnvVar ("Username")** fetches a value from the system environment, in this case the value of the environment variable *Username*. Functions may have any number of parameters, including zero:  
**GetMsVersionInfo**  
 On a win7 system, this function call yields the value "6.1" (not as with a variable this values has to be produced at every call again).
- A value can be constructed by an additive expression, where string values and partial expressions are concatenated - theoretically "plus" can be seen as a function of two parameters:  
*\$Home\$ + "\mail"*

(More on this in Section 7.3)

There is no analogous way of using string expressions in the secondary sections. They follow there domain specific syntax. e.g. for copying commands similar to the windows command line copy command. Up to this moment it is no escape syntax implemented for transporting primary section logic into secondary sections.

The only way to transport string values into secondary sections is the use of the names of variables and constants as value container in these sections. Lets have a closer look at the variables and constants of a *opsi-winst* script:

## Chapter 6

# Definition and Use of Variables and Constants in a opsi-winst Script

### 6.1 General

In a *opsi-winst* script, variables and constants appear as "words", that are interpreted by *opsi-winst* and "contain" values. "Words" are sequences of characters consisting of letters, numbers and some special characters (in particular ".", "-", "\_", "\$", "%"), but not blanks, but no brackets, parentheses, or operator signs ("+").

*opsi-winst* variables and constants are not case-sensitive.

There exist the following types of variables or constants:

- Global text constants, shortly constants, contain values which are present by the *opsi-winst* program and cannot be changed in a script. Before interpreting the script *opsi-winst* replaces each occurrence of the pure constant name with its value in the whole script (textual substitution).  
An example will make this clear:  
The constant `%ScriptPath%` is the predefined name of the location where *opsi-winst* found and read the script that it just executes. This location may be, e.g., `p:\product`. Then we have to write `"%ScriptPath%"` in the script when we want do get the value `"p:\product"`.  
– observe the citations marks which include the constant delimiter.
- Text or String variables, shortly variables, have an appearance very much like any (String) variables in a common programming language. They must be declared by a `DefVar` statement before they can be used. In primary sections, values can be assigned to variables (once ore more times). They can be used as elements in composed expressions (like addition of strings) or as function arguments.  
But they freeze in a secondary section to a phenomenon that behaves like a constant. There, they appear as a non-syntactical foreign element. Their value is fixed and is inserted by textual substitution for their pure names (when a section is called, whereas the textual substitution for real constants take place before starting the execution of the whole script).
- Stringlist variables are declared by a `DefStringList` statement. In primary sections they can be used for many purposes, e.g. collecting strings, manipulating strings, building sections.

In detail:

## 6.2 Global Text Constants

Scripts shall work in a different contexts without manual changes. The contexts can be characterized by system values as OS version or certain paths. *opsi-winst* introduces such values as constants into the script.

### 6.2.1 Usage

The fundamental characteristics of a text constant is the way how the values which it represents come into the script interpretation process:

The name of the constant, that is the pure sequences of chars, is substituted by its fixed value in the whole script before starting the script execution.

The replacement does not take into account any syntactical context in which the name possibly occur (exactly like with variables in secondary sections).

### 6.2.2 Example

*opsi-winst* implements constants `%ScriptPath%` for the location of the momentarily interpreted script and `%System%` for the name of the windows system directory. The following (Files) subsection defines a command that copies all files from the script directory to the windows system directory:

```
[files_do_my_copying]
copy "%ScriptPath%\system\*.*" "%System%"
```

At this moment the following constants are implemented:

### 6.2.3 System paths

#### 6.2.3.1 Base system directories

```
%ProgramFilesDir%: c:\program files
%ProgramFiles32Dir%: c:\Program Files (x86)
%ProgramFiles64Dir%: c:\program files
%ProgramFilesSysnativeDir%: c:\program files
%Systemroot%: c:\windows
%System%: c:\windows\system32
%Systemdrive%: c:
%ProfileDir%: c:\Documents and Settings
```

#### 6.2.3.2 Common (AllUsers) directories

```
%AllUsersProfileDir% or %CommonProfileDir%: c:\Documents and Settings\All Users
%CommonStartMenuPath% or %CommonStartmenuDir%: c:\Documents and Settings\All Users\Startmenu
%CommonAppdataDir%: c:\Documents and Settings\All Users\Application Data
%CommonDesktopDir%
%CommonStartupDir%
%CommonProgramsDir%
```

### 6.2.3.3 Current (logged in or usercontext) user directories

`%AppdataDir%` or `%CurrentAppdataDir%` : `c:\Documents and Settings\%USERNAME%\Application Data`

`%CurrentStartmenuDir%`

`%CurrentDesktopDir%`

`%CurrentStartupDir%`

`%CurrentProgramsDir%`

`%CurrentSendToDir%`

`%CurrentProfileDir%` //since 4.11.2.1

### 6.2.3.4 /AllNtUserProfiles directory constants

In *Files* sections that are called with option `/AllNtUserProfiles` there is a pseudo variable

`%UserProfileDir%`

When the section is executed for each user that exists on a work station this variable represents the name of the profile directory of the user just treated.

`%CurrentProfileDir%` // since 4.11.2.1

may be used instead of the older `%UserProfileDir%` in order to have Files-sections which may be used also for *userLoginScripts*.

## 6.2.4 opsi-winst Paths

`%ScriptPath%` or `%ScriptDir%` : represents the path of the current *opsi-winst* script (without closing backslash). Using this variable we can build path and file names in scripts that are relative to the location of the script. So, everything can be copied, called from the new place, and all works as before.

`%ScriptDrive%` : the drive where the just executed *opsi-winst* script is located (including the colon).

`%WinstDir%` : The location (without closing backslash) of the running *opsi-winst*.

`%WinstVersion%` : Version string of the running *winst*.

`%Logfile%` : The name of the logfile which *opsi-winst* is using.

Example:

The code:

```
comment "Testing: "
message "Testing constants: "+"%"+ "winstversion" +"%"
set $ConstTest$ = "%WinstVersion%"
set $InterestingFile$ = "%winstdir%\winst.exe"
if not (FileExists($InterestingFile$))
    set $InterestingFile$ = "%winstdir%\winst32.exe"
endif
set $INST_Resultlist$ = getFileInfoMap($InterestingFile$)
set $CompValue$ = getValue("file version with dots", $INST_Resultlist$ )
if ($ConstTest$ = $CompValue$)
    comment "passed"
else
    set $TestResult$ = "not o.k."
    LogWarning "failed"
endif
```

results to the following log:

```

comment: Testing:
message Testing constants: %winstversion%

Set $ConstTest$ = "4.10.8.3"
  The value of the variable "$ConstTest$" is now: "4.10.8.3"

Set $InterestingFile$ = "N:\develop\delphi\winst32\trunk\winst.exe"
  The value of the variable "$InterestingFile$" is now: "N:\develop\delphi\winst32\trunk\winst.
  exe"

If
  Starting query if file exist ...
  FileExists($InterestingFile$) <<< result true
  not (FileExists($InterestingFile$)) <<< result false
Then
EndIf

Set $INST_Resultlist$ = getFileInfoMap($InterestingFile$)
  retrieving strings from getFileInfoMap [switch to loglevel 7 for debugging]

Set $CompValue$ = getValue("file version with dots", $INST_Resultlist$ )
  retrieving strings from $INST_Resultlist$ [switch to loglevel 7 for debugging]
  The value of the variable "$CompValue$" is now: "4.10.8.3"

If
  $ConstTest$ = $CompValue$ <<< result true
  ($ConstTest$ = $CompValue$) <<< result true
Then
  comment: passed

Else
EndIf

```

## 6.2.5 Network Information

**%Host%** : (Deprecated) The value of a environmental variable host (traditionally meaning the opsi server name, not to confuse with %HostID% (meaning the client network name).

**%PCName%**: The value of the environmental variable PCName, when existing. Otherwise the value of the environmental variable computername. (Should be the netbios name of the PC)

**%IPName%** : The dns name of the pc. Usually identical with the netbios name and therefore with %PCName% besides that the netbios names uses to be uppercase.

**%Username%** : Name of the logged in user.

## 6.2.6 Data for and from opsi service

**%HostID%** : Should be the fully qualified domain name of the opsi client as it is supplied from the command line or otherwise.

**%opsiserviceURL%** : The (usually https://) URL of the opsi service.

**%opsiServer%** : The server name derived from the %opsiserviceURL%.

**%opsiserviceUser%** : The user ID for which there is a connection to the opsi service.

**%opsiservicePassword%** : The user password used for the connection to the opsi service. The password is eliminated when logging by the standard *opsi-winst* logging functions.

**%installingProdName%**: The *productid* of the product that is actually installed via call by the opsi-service. Empty if the Script ist not started by the opsi-service.

**%installingProdVersion%**: A String combined from `<productversion>-<packageversion>` for the product that is actually installed via call by the opsi-service. Empty if the Script ist not started by the opsi-service.

**%installingProduct%** : (Deprecated) The name (productId) of the product for which the service has called the running script. In case that there the script is not run via the service the String is empty.

## 6.3 String (or Text) Variables

### 6.3.1 Declaration

String variables must be declared before they can be used. The syntax for the declaration reads

```
DefVar <variable name>
```

e.g.

```
DefVar $MsVersion$
```

Explanation:

- Variable names do not necessarily start or end with a dollar sign, but this is recommended as a convention to understand their functioning in secondary sections.
- Variables can only be declared in primary sections (Initial section, Actions section sub sections and ProfileActions).
- The declaration should not depend on a condition. That is it should not be placed into a branch of an if – else statement. Otherwise, it could happen that the DefVar statement is not executed for a variable, but an evaluation of the variable is tried in some if clause (such producing a syntax error). The variables are initialized with an empty string ("").

Recommendation:

- The first and last letter of the name should be \$
- Define all variables at the beginning of the script

### 6.3.2 Value Assignment

As it is appropriate for a variable, it can take on one value resp. a series of values while a script is progressing. The values are assigned by statements with syntax

```
Set <Variablename> = <Value>
```

<Value> means any (String valued) expression.

Examples (For Examples see Section 7.3):

```
Set $OS$ = GetOS
Set $NTVersion$ = "nicht bestimmt"

if $OS$ = "Windows_NT"
  Set $NTVersion$ = GetNTVersion
endif
```

```

DefVar $Home$
Set $Home$ = "n:\home\user name"
DefVar $MailLocation$
Set $MailLocation$ = $Home$ + "\mail"

```

### 6.3.3 Use of variables in String expressions

In primary sections of a *opsi-winst* script, a variable "holds" a value. When it is declared it is initialized with the empty String "". When a new value is assigned to it via the `set` command, it represents this value.

In a primary section a variable can replace any String expression resp. can be a component of a String expression, e.g.

```

Set $MailLocation$ = $Home$ + "\mail"

```

In a primary section the variable name denotes an object that represents a string, If we add the variable we mean that the underlying string shall be added somehow.

This representational chain is shortcut in a secondary section. Just the variable name now stands for the string.

### 6.3.4 Secondary vs. primary sections

When a secondary section is loaded and *opsi-winst* starts its interpretation the sequence of chars of a variable name is directly replaced by the value of the variable.

Example:

A copy command in a files section shall copy a file to

```
"n:\home\user name\mail\backup"
```

kopiert werden.

We first set `$MailLocation$` to the directory above it:

```

DefVar $Home$
DevVar $MailLocation$
Set $Home$ = "n:\home\user name"
Set $MailLocation$ = $Home$ + "\mail"

```

`$MailLocation$` is now holding

```
"n:\home\user name\mail"
```

In a primary section we may now express the directory

```
"n:\home\user name\mail\backup"
```

by

```
$MailLocation$ + "\backup"
```

The same directory has to be designated in a secondary section as:

```
"$MailLocation$\backup"
```

A fundamental difference between the thinking of variables in primary vs. secondary sections is that, in a primary section, we can form an assignment expression like

```
$MailLocation$ = $MailLocation$ + "\backup"
```

As usual, this means that `$MailLocation$` first has some initial value and takes on a new value by adding some string to the initial value. The reference from the variable is dynamic, and may have a history.

In a secondary section any such expression would be worthless (and eventually wrong), since `$MailLocation$` is bound to be replaced by some fixed string (at all occurrences virtually in the same moment).

## 6.4 Stringlist Variables

Variables for string lists must be declared in a `DefStringList` statement, e.g.

```
DefStringList SMBMounts
```

A string list can serve e.g. as container for the captured output of a shell program. The collected strings can be manipulated in a lot of ways. In detail this will be treated in the section on string list processing (see Section 7.4).



### Caution

Wenn (geschachtelte) Sub-Sektionen in externe Dateien ausgelagert werden, müssen die aufgerufenen Sekundären Sektionen üblicherweise in der Datei untergebracht werden, aus der sie aufgerufen werden. Je nach verwendeter Komplexität des Syntax müssen sie evtl. **zusätzlich** auch in der Hauptdatei untergebracht werden.

## Chapter 7

# Syntax and Meaning of Primary Sections of a opsi-winst Script

As shortly presented in chapter 4 the Actions section of a script can be regarded as a the main method of the *opsi-winst* script and describes the global processing sequence. It may call subroutines - the Sub sections which may then recursively call Sub sections themselves.

The following sections explain syntax and use of the primary sections of a *opsi-winst* script.

### 7.1 Primary Sections

There are possibly three kinds of primary sections in a script

- an **Initial** section (may be omitted),
- an **Action** section,
- any number of **Sub** sections
- an **ProfileActions** section

**Initial** and **Action** section are syntactically equivalent (but **Initial** has to keep the first place). By convention, in the **Initial** section some parametrizations of the script execution (e.g. the loglevel) are made. The **Action** section can be regarded as the main program in a *opsi-winst* script. It contains the sequence of actions that are controlled by the script.

Sub sections are as well syntactically equivalent. But they are a called from the **Action** section. Then, they can call themselves **Sub** sections.

A **Sub** section is determined by creating a name that begins with "Sub", e.g. `Sub_InstallBrowser`. By writing its name in the **Action** section we produce a call to the **Sub** section. The meaning of this call is defined by the content of the section in the script that begins with the bracketed name, in the example `[Sub_InstallBrowser]`

---

#### Note

Subsections of second and higher order cannot host internal sections. Instead, their procedure calls must refer to sections defined in the main script file or defined as external sections (cf. Section [7.10](#)).

---



#### Caution

If (nested) sub sections are externalized to external files, the called sections has to be in that file where they are called from. According to the complexity of the script they may sometimes have to be placed **also** in the main file.

---

A `ProfileActions` section at a normal installation script may be used as a sub section with a special syntax. In a `userLoginScript` this section will be used as script start (instead of `Actions`). See chapter *User Profile Management* at the opsi-manual and Section 7.8.

## 7.2 Parametrizing opsi-winst

Typical entries of an Initial section set some the *opsi-winst* execution attributes. The following example shows how error responses may be configured:

### 7.2.1 Example

```
[Initial]
SetLogLevel=5
ExitOnError=false
ScriptErrorMessages=on
TraceMode=off
```

This means that:

- logging level is set to 5
- when an error occurs *opsi-winst* shall try to continue script execution
- if a script syntax error occurs it shall be communicated (this will be in a special window)
- we don't want to activate the trace mode for script execution (which would mean that we are asked after each program step if we want to continue).

The above values are the default values, *opsi-winst* will assume them if these statements are missing.

To the details of syntax and meaning:

### 7.2.2 Specification of Logging Level



#### Caution

The old function `LogLevel=` is deprecated since *opsi-winst* version 4.10.3. For backward compatibility reasons `Loglevels` ste by this old function will be increased by 4 before they are used.

There are two syntactical variants for specifying the logging level:

`SetLogLevel = <number>` `SetLogLevel = <String expression>` I.e. the number can be given as an integer value or as a string expression (cf. section 6.3). In the second case, *opsi-winst* tries to evaluate the string expression as a number. There exist ten levels from 0 up to 9.

Es gibt zwei ähnliche Varianten, um den Loglevel zu spezifizieren:

`SetLogLevel = <number>`

`SetLogLevel = <String expression>`

I.e. the number can be given as an integer value or as a string expression (cf. Section 7.3). In the second case, *opsi-winst* tries to evaluate the string expression as a number.

There exist ten levels from 0 up to 9.

```

0 = nothing (absolute nothing)
1 = essential ("essential information")
2 = critical (unexpected errors that may cause a program abort)
3 = error (Errors that don't will abort the running program)
4 = warning (you should have a look at this)
5 = notice (Important statements to the program flow)
6 = info (Additional Infos)
7 = debug (important debug messages)
8 = debug2 (a lot more debug informations and data)
9 = confidential (passwords and other security relevant data)

```

### 7.2.3 Required opsi-winst Version

The statement

```
requiredWinstVersion <RELATION SYMBOL> <NUMBER STRING>
```

e.g.

```
requiredWinstVersion >= "4.3"
```

makes *opsi-winst* check if the desired version state is given. Otherwise an error message window pops up.

This feature exists since *opsi-winst* version 4.3. For an earlier version, the statement is unknown, and the statement itself is a syntactical error which will be indicated by syntax error window (cf. the following section). Therefore the statement can be used independently of the currently used *opsi-winst* version as long as the required version is at least version 4.3.

### 7.2.4 Reacting on Errors

There are two kinds of errors which are treated in different ways:

1. illegal statements which cannot be interpreted by *opsi-winst* (syntactical errors),
2. failing statements which cannot be executed because of external, objective reasons (execution errors).

In principal, syntactical errors are indicated by a pop up window for immediate correction, execution errors are logged in a log file to be analysed later.

The behaviour of *opsi-winst* when it recognizes a syntactical error is defined by the configuration statement

- **ScriptErrorMessages** = <boolean value>

If the value is true (default), syntactical errors trigger a pop up window with some informations on the error. This kind of errors is not recorded in the log file. The log file shall keep informations on the real execution of a syntactical correct script.

The boolean value may be true or false. Delimiters on or off can be used as well.

There two configuration options for execution errors.

- **ExitOnError** = <boolean value>

This statement defines if the script execution shall terminate when an error occurs. If the value is true or yes the program will stop execution, otherwise errors are just logged (default).

- **TraceMode** = <boolean value>

In TraceMode (default false) every log file entry will additionally be shown in message window with an O.K. button.

## 7.2.5 Staying On Top

- `StayOnTop = <Wahrheitswert>`

With `StayOnTop = true` (or `= on`) we request, that - in batch mode - the *opsi-winst* window be on top on the windows which share the screen. That means it should be visible in the "foreground" as long as no other window having the same status wins.



### Caution

According to the system manual the value cannot be changed while the program is running. But it seems that we can give a new value to it once.

`StayOnTop` has default `false` in order to avoid that some other process raises an error message which eventually can not be seen if *opsi-winst* keeps staying on top.

## 7.3 String Expressions, String Values, and String Functions

A String expression can be

- an elementary String value
- a nested String value
- a String variable
- the concatenation of other String expressions
- a String valued function call

### 7.3.1 Elementary String Values

An elementary String value is any sequence of characters that is enclosed in double or single citations marks, formally:

`"<sequence of characters>"`

or

`'<sequence of characters>'`

Example:

```
DefVar $ExampleString$
Set $ExampleString$ = "my Text"
```

### 7.3.2 Strings in Strings (Nested String Values)

If the sequence of chars itself contains citation marks we have to use the other kind of citation marks to enclose it:

```
DefVar $citation$
Set $citation$ = 'he said "Yes"'
```

If the sequence of chars is containing both kinds of citation marks we must use the following special expression:

`EscapeString: <sequence of characters>`

E.g. we can write:

```
DefVar $Meta_citation$
Set $Meta_citation$ = EscapeString: Set $citation$ = 'he said "Yes"'
```

Then the variable `$Meta_citation$` will exactly contain the complete sequence of chars that follows the colon after "EscapeString" (including the blank). Such, `$Meta_citation$` will contain the complete statement: `Set $citation$ = 'he said "Yes"'`

### 7.3.3 String Concatenation

String concatenation is written using the addition sign ("+")

```
<String expression> + <String expression>
```

Example:

```
DefVar $String1$
DefVar $String2$
DefVar $String3$
DefVar $String4$
Set $String1$ = "my text"
Set $String2$ = "and"
Set $String3$ = "your text"
Set $String4$ = $String1$ + " " + $String2$ + " " + $String3$
```

`$String4$` then has value "my text and your text".

### 7.3.4 String Variables

A String variable in a primary section "contains" a String value. In an String expression, it can always substitute an elementary string. For how to define and set String variables cf. Section 6.3.

The following sections present the variety of string functions.

### 7.3.5 String Functions which Return the OS Type

- GetOS

The function tells which type of operating system is running.

We recommend to use `GetMsVersionInfo`.

`GetOS`` returns one of the following values:

"Windows\_16"

"Windows\_95" (including Windows 98 and ME)

"Windows\_NT" (including Windows 2000 and XP)

"Linux"

- GetNtVersion

Deprecated - please use `GetMsVersionInfo`.

A Windows NT operating system is characterized by a the Windows type number and a subtype number. `GetNtVersion` returns the precise subtype name. Possible values are

"NT3"

"NT4"

"Win2k" (Windows 5.0)

"WinXP" (Windows 5.1)

"Windows Vista" (Windows 6)

If the NT operating system has higher versions as 6 or there are version not explicitly known the function returns "Win NT" and the complete version number (5.2, ... resp. 6.0 ..) . E.g. for Windows Server 2003 R2 Enterprise

Edition, we get

"Win NT 5.2"

If the operating system is no Windows NT system the function returns the error value

"No OS of Windows NT type"

- **GetMsVersionInfo**  
returns for systems of type Windows NT the Microsoft version info as indicated by the API, e.g. a Windows XP system produces the result  
"5.1"

Table 7.1: Windows Versions

<b>GetMsVersionInfo</b>	<b>Windows Version</b>
5.0	Windows 2000
5.1	Windows XP (Home, Prof)
5.2	XP 64 Bit, 2003, Home Server, 2003 R2
6.0	Vista, 2008
6.1	Windows 7, 2008 R2

see also `GetMsVersionMap`

- **GetSystemType**  
checks the installed Windows OS if it can be assumed that the system is 64 Bit. In this case the value is *64 Bit System* otherwise *x86 System*.

### 7.3.6 String Functions for Retrieving Environment or Command Line Data

The function reads and returns the momentary value of a system environment variable.

E.g., we can retrieve which user is logged in by `EnvVar ("Username")`. `ParamStr` The function passes the the parameter string of the *opsi-winst* command line i.e. the command line parameter which is indicated by `/parameter`. If there is no such parameter `ParamStr` returns the empty string. `GetLastExitCode` returns the exit code (also called `ErroLevel`) of the last Winbatch call. `GetUserSID(<Windows Username>)` returns the SID for a given user (possibly with domain prefix in the form `DOMAIN\USER`).

- **EnvVar (<string>)**  
The function reads and returns the momentary value of a system environment variable. E.g., we can retrieve which user is logged in by `EnvVar ("Username")`.
- **ParamStr**  
The function passes the the parameter string of the *opsi-winst* command line i.e. the command line parameter which is indicated by `/parameter`. If there is no such parameter `ParamStr` returns the empty string.
- **GetLastExitCode**  
returns the exit code (also called `ErroLevel`) of the last Winbatch call.
- **GetUserSID(<Windows Username>)**  
returns the SID for a given user (possibly with domain prefix in the form `DOMAIN\USER`).
- **GetUsercontext**  
returns the string which was given to the *opsi-winst* by the optional parameter `/usercontext`. IF this parameter was not used the returned string is empty.

### 7.3.7 Reading Values from the Windows Registry and Transforming Values into Registry Format

- `GetRegistryStringValue (<string>)`  
tries to interpret the passed String value as an expression of format `[KEY] X`  
Then, the function tries to open the registry key `KEY`, and, in case it succeeds, to read and return the String value that belongs to the registry variable name `X`.

E.g.

```
GetRegistryStringValue (" [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
Winlogon] Shell")
```

usually yields "Explorer.exe", the default Windows shell program.

If there is no registry key `KEY` or the variable `X` does not exist the function produces a warning message in the log file and returns the empty string.

For example: If we made a *standard entry* with the value `standard entry` at the key `HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org\opsi-winst-test\test-4.0`, we will get with

```
Set $CompValue$ = GetRegistryStringValue32 (" [HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org\opsi-winst-
test\test-4.0] ")
```

the following log:

```
Registry started with redirection (32 Bit)
Registry key [HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org\opsi-winst-test\test-4.0] opened
Key closed
The value of the variable "$CompValue$" is now: "standard entry"
```

- `GetRegistryStringValue32(<string>)` → see [Chapter 64 Bit](#)
- `GetRegistryStringValue64(<string>)` → see [Chapter 64 Bit](#)
- `GetRegistryStringValueSysNative(<string>)` → see [Chapter 64 Bit](#)
- `RegString(<string>)`  
is useful for transforming path names into the format which is used in the Windows registry. That is, any backslash is duplicated. E. g.,

```
RegString ("c:\windows\system\")
```

yields

```
"c:|\windows|system|"
```

### 7.3.8 Reading Values from ini files

For historical reasons, there are three functions for reading values from configuration files which have ini file format. Since opsi 3.0 the specific product properties are retrieved from the opsi configuration demon (that may fetch it from a configuration file or from any other backend data container).

In detail:

Ini file format means that the file is a text file and is composed of "sections" each containing key value pairs:

```
[section1]
Varname1=Value1
Varname2=Value2
...
[section2]
...
```

The most general function reads the value belonging to some key in some section of some ini file. Any parameter can be given as an arbitrary String expression:

- `GetValueFromInifile (<FILE>, <SECTION>, <KEY>, <DEFAULTVALUE>)`  
The function tries to open the ini file `FILE`, retrieve the requested `SECTION` and find the value belonging to the specified `KEY` which the function will return. If any of these operations fail `DEFAULTVALUE` is returned.

The second function borrows its syntax from the ini file format itself, and may sometimes be easier to use. But since this syntax turns complicated in more general circumstances it is deprecated. The syntax reads:

- `GetIni (<Stringausdruck> [ <character sequence> ] <character sequence> )`  
(Deprecated) The `<String expression>` is interpreted as file name, the first `<character sequence>` as section name, the second as key name.

### 7.3.9 Reading Product Properties

- `GetProductProperty (<PropertyName>, <DefaultValue>)`  
where `$PropertyName$` and `$DefaultValue$` are String expressions. If *opsi-winst* is connected to the opsi configuration service the product property is retrieved from the service. If the *opsi-winst* is not connected to the service or for other reasons the the call fails, the given `<DefaultValue>` will be returned.

The product properties can be used to configure variants of an installation.

E.g. the opsi UltraVNC network viewer installation may be configured using the options

- `viewer = <yes> | <no>`
- `policy = <factory_default> |`

The installation script branches according to the chosen values for these options which can be retrieved by

```
GetProductProperty("viewer", "yes")
GetProductProperty("policy", "factory_default")
```

- `IniVar(<PropertyName>)`  
(deprecated: use `GetProductProperty`)

### 7.3.10 Retrieving Data from etc/hosts

- `GetHostsName(<string>)`  
returns the host name to a given IP address as it is declared in the local hosts file. If the operating system is "Windows\_NT" (according to environment variable `OS`) "%systemroot%\system32\drivers\etc\" is assumed as host file location, otherwise "C:\Windows\".
- `GetHostsAddr(<string>)`  
tells the IP address to a given host or alias name.

### 7.3.11 String processing

- **ExtractFilePath(<string>)**  
interprets the passed String value as file or path name and returns the path part (the string up to the last "\", including it).
- **StringSplit (`STRINGWERT1, STRINGWERT2, INDEX)`**  
(deprecated: use `splitString / takestring`)
- **takeString(<index>,<list>)**  
returns from a string list <list> the string with the index <index>.  
Often used in combination with `splitstring`: `takeString(<index>, splitString(<string1>, <string2>)`  
(see also Section 7.4).  
The result is produced by slicing <string1> where each slice is delimited by an occurrence of <string2>, and then taking the slice with index <index> (where counting starts with 0).

Example:

```
takeString(3, splitString ("\\server\share\directory", "\"))
```

returns *"share"*,

the given string splitted at "\" returns the string list:

Index 0 - "" (empty string), because there is nothing before the first "\"

Index 1 - "" (empty string), because there is nothing before the second "\"

Index 2 - "server"

Index 3 - "share"

Index 4 - "directory"

`takestring` counts downward, if the index is negative, starting with the number of elements. Therefore,

```
takestring(-1, $list1$)
```

denotes the last element of String list `$list1$`.

- **SubstringBefore(<string1>, <string2>)**  
(deprecated: use `splitString / takestring`) yields the sequence of characters of `stringValue1` up to the beginning of `stringValue2`.  
Example:

```
SubstringBefore ("C:\programme\staroffice\program\soffice.exe", "\program\soffice.exe")
```

returns *"C:\programme\staroffice"*.

- **takeFirstStringContaining(<list>,<search string>)**  
returns the first string from <list> which contains <search string>.  
Returns an empty string if no matching string is found.
- **Trim(<string>)**  
cuts leading and trailing white space from <string>.
- **lower(<string>)**  
returns <string> with lower case.
- **unquote(<string>,<quote-string>) //since 4.11.2.1**  
If <string> is quoted by <quote-string> it returns the unquoted <string>  
From <quote-string> is only the first char used, leading whitespaces are ignored.
- **HexStrToDecStr(<string>)**  
returns the decimal representation of the input string if this was the hexadecimal representation of an integer.  
Leading chars like *0x* or *\$* will be ignored. In case of a converting error the function returns an empty string.

- `DecStrToHexStr(<string>)` returns the hexadecimal representation of the input string if this was the decimal representation of an integer. In case of a converting error the function returns a empty string.
- `base64EncodeStr(<string>)`  
returns the base64 encoded value of <string>.
- `base64DecodeStr(<string>)`  
returns the base64 decoded value of <string>.

### 7.3.12 Weitere String-Funktionen

- `RandomStr`  
returns a random String of length 10 where upper case letters, lower case letters and digits are mixed (for creating passwords). More exactly: 2 lower case chars, 2 upper case chars, 2 special chars and 4 digits. The possible special chars are:  
`!,$, (, ), *, +, /, ;, =, ?, [, ], {, }, ^, ~, $, °`
- `CompareDotSeparatedNumbers(<string1>, <string2>)`  
compares two strings of the form <number>.<number>[.<number>[.<number>]]  
It returns "0" if the strings are equal, "1" if <string1> is higher and "-1" if <string1> is lower than <string2>.

Example:

The Code:

```

comment "Testing: "
message "CompareDotSeparatedNumbers"
set $string1$ = "1.2.3.4.5"
set $string2$ = "1.2.3.4.5"
set $ConstTest$ = "0"
set $CompValue$ = CompareDotSeparatedNumbers($string1$, $string2$)
if ($ConstTest$ = $CompValue$)
    comment "passed"
    comment "$string1$" is equal to "$string2$"
else
    set $TestResult$ = "not o.k."
    LogWarning "failed"
endif

set $string1$ = "1.2.31.4.5"
set $string2$ = "1.2.13.4.5"
set $ConstTest$ = "1"
set $CompValue$ = CompareDotSeparatedNumbers($string1$, $string2$)
if ($ConstTest$ = $CompValue$)
    comment "passed"
    comment "$string1$" is higher then "$string2$"
else
    set $TestResult$ = "not o.k."
    LogWarning "failed"
endif

set $string1$ = "1.2.3.4.5"
set $string2$ = "1.2.13.4.5"
set $ConstTest$ = "-1"
set $CompValue$ = CompareDotSeparatedNumbers($string1$, $string2$)
if ($ConstTest$ = $CompValue$)
    comment "passed"
    comment "$string1$" is lower then "$string2$"

```

```

else
    set $TestResult$ = "not o.k."
    LogWarning "failed"
endif

comment ""
comment "-----"
comment "Testing: "
message "CompareDotSeparatedStrings"
set $string1$ = "1.a.b.c.3"
set $string2$ = "1.a.b.c.3"
set $ConstTest$ = "0"
set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
if ($ConstTest$ = $CompValue$)
    comment "passed"
    comment $string1$+" is equal to "+$string2$
else
    set $TestResult$ = "not o.k."
    LogWarning "failed"
endif

```

leads to the following log:

```

comment: Testing:
message CompareDotSeparatedNumbers

Set $string1$ = "1.2.3.4.5"
  The value of the variable "$string1$" is now: "1.2.3.4.5"

Set $string2$ = "1.2.3.4.5"
  The value of the variable "$string2$" is now: "1.2.3.4.5"

Set $ConstTest$ = "0"
  The value of the variable "$ConstTest$" is now: "0"

Set $CompValue$ = CompareDotSeparatedNumbers($string1$, $string2$)
  The value of the variable "$CompValue$" is now: "0"

If
  $ConstTest$ = $CompValue$ <<< result true
  ($ConstTest$ = $CompValue$) <<< result true
Then
  comment: passed
  comment: 1.2.3.4.5 is equal to 1.2.3.4.5

Else
EndIf

Set $string1$ = "1.2.31.4.5"
  The value of the variable "$string1$" is now: "1.2.31.4.5"

Set $string2$ = "1.2.13.4.5"
  The value of the variable "$string2$" is now: "1.2.13.4.5"

Set $ConstTest$ = "1"
  The value of the variable "$ConstTest$" is now: "1"

```

```

Set $CompValue$ = CompareDotSeparatedNumbers($string1$, $string2$)
The value of the variable "$CompValue$" is now: "1"

If
  $ConstTest$ = $CompValue$ <<< result true
  ($ConstTest$ = $CompValue$) <<< result true
Then
  comment: passed
  comment: 1.2.31.4.5 is higher then 1.2.13.4.5

Else
EndIf

Set $string1$ = "1.2.3.4.5"
The value of the variable "$string1$" is now: "1.2.3.4.5"

Set $string2$ = "1.2.13.4.5"
The value of the variable "$string2$" is now: "1.2.13.4.5"

Set $ConstTest$ = "-1"
The value of the variable "$ConstTest$" is now: "-1"

Set $CompValue$ = CompareDotSeparatedNumbers($string1$, $string2$)
The value of the variable "$CompValue$" is now: "-1"

If
  $ConstTest$ = $CompValue$ <<< result true
  ($ConstTest$ = $CompValue$) <<< result true
Then
  comment: passed
  comment: 1.2.3.4.5 is lower then 1.2.13.4.5

Else
EndIf

```

- `CompareDotSeparatedStrings (<string1>, <string2>)`  
 compares two strings of the form `<string>.<string>[.<string>[.<string>]]`  
 It returns "0" if the strings are equal, "1" if `<string1>` is higher and "-1" if `<string1>` is lower than `<string2>`. The function is not case sensitive.

Example:

The Code:

```

comment "Testing: "
message "CompareDotSeparatedStrings"
set $string1$ = "1.a.b.c.3"
set $string2$ = "1.a.b.c.3"
set $ConstTest$ = "0"
set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
if ($ConstTest$ = $CompValue$)
  comment "passed"
  comment "$string1$+" is equal to "+$string2$"
else
  set $TestResult$ = "not o.k."
  LogWarning "failed"
endif

```

```

set $string1$ = "1.a.b.c.3"
set $string2$ = "1.A.B.C.3"
set $ConstTest$ = "0"
set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
if ($ConstTest$ = $CompValue$)
    comment "passed"
    comment $string1$+" is equal to "+$string2$
else
    set $TestResult$ = "not o.k."
    LogWarning "failed"
endif

set $string1$ = "1.a.cb.c.3"
set $string2$ = "1.a.b.c.3"
set $ConstTest$ = "1"
set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
if ($ConstTest$ = $CompValue$)
    comment "passed"
    comment $string1$+" is higher then "+$string2$
else
    set $TestResult$ = "not o.k."
    LogWarning "failed"
endif

set $string1$ = "1.a.ab.c.3"
set $string2$ = "1.a.b.c.3"
set $ConstTest$ = "-1"
set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
if ($ConstTest$ = $CompValue$)
    comment "passed"
    comment $string1$+" is lower then "+$string2$
else
    set $TestResult$ = "not o.k."
    LogWarning "failed"
endif

set $string1$ = "1.2.13.4.5"
set $string2$ = "1.2.3.4.5"
set $ConstTest$ = "-1"
set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
if ($ConstTest$ = $CompValue$)
    comment "passed"
    comment $string1$+" is lower then "+$string2$
    comment "using CompareDotSeparatedStrings give wrong results on numbers"
else
    set $TestResult$ = "not o.k."
    LogWarning "failed"
endif

set $string1$ = "1.2.3.4.5"
set $string2$ = "1.2.13.4.5"
set $ConstTest$ = "1"
set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
if ($ConstTest$ = $CompValue$)
    comment "passed"
    comment $string1$+" is higher then "+$string2$

```

```

        comment "using CompareDotSeparatedStrings give wrong results on numbers"
    else
        set $TestResult$ = "not o.k."
        LogWarning "failed"
    endif

```

leads to the following log:

```

comment: Testing:
message CompareDotSeparatedStrings

Set $string1$ = "1.a.b.c.3"
  The value of the variable "$string1$" is now: "1.a.b.c.3"

Set $string2$ = "1.a.b.c.3"
  The value of the variable "$string2$" is now: "1.a.b.c.3"

Set $ConstTest$ = "0"
  The value of the variable "$ConstTest$" is now: "0"

Set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
  The value of the variable "$CompValue$" is now: "0"

If
  $ConstTest$ = $CompValue$ <<< result true
  ($ConstTest$ = $CompValue$) <<< result true
Then
  comment: passed
  comment: 1.a.b.c.3 is equal to 1.a.b.c.3

Else
EndIf

Set $string1$ = "1.a.b.c.3"
  The value of the variable "$string1$" is now: "1.a.b.c.3"

Set $string2$ = "1.A.B.C.3"
  The value of the variable "$string2$" is now: "1.A.B.C.3"

Set $ConstTest$ = "0"
  The value of the variable "$ConstTest$" is now: "0"

Set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
  The value of the variable "$CompValue$" is now: "0"

If
  $ConstTest$ = $CompValue$ <<< result true
  ($ConstTest$ = $CompValue$) <<< result true
Then
  comment: passed
  comment: 1.a.b.c.3 is equal to 1.A.B.C.3

Else
EndIf

Set $string1$ = "1.a.cb.c.3"
  The value of the variable "$string1$" is now: "1.a.cb.c.3"

```

```
Set $string2$ = "1.a.b.c.3"
  The value of the variable "$string2$" is now: "1.a.b.c.3"

Set $ConstTest$ = "1"
  The value of the variable "$ConstTest$" is now: "1"

Set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
  The value of the variable "$CompValue$" is now: "1"

If
  $ConstTest$ = $CompValue$ <<< result true
  ($ConstTest$ = $CompValue$) <<< result true
Then
  comment: passed
  comment: 1.a.cb.c.3 is higher then 1.a.b.c.3

Else
EndIf

Set $string1$ = "1.a.ab.c.3"
  The value of the variable "$string1$" is now: "1.a.ab.c.3"

Set $string2$ = "1.a.b.c.3"
  The value of the variable "$string2$" is now: "1.a.b.c.3"

Set $ConstTest$ = "-1"
  The value of the variable "$ConstTest$" is now: "-1"

Set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
  The value of the variable "$CompValue$" is now: "-1"

If
  $ConstTest$ = $CompValue$ <<< result true
  ($ConstTest$ = $CompValue$) <<< result true
Then
  comment: passed
  comment: 1.a.ab.c.3 is lower then 1.a.b.c.3

Else
EndIf

Set $string1$ = "1.2.13.4.5"
  The value of the variable "$string1$" is now: "1.2.13.4.5"

Set $string2$ = "1.2.3.4.5"
  The value of the variable "$string2$" is now: "1.2.3.4.5"

Set $ConstTest$ = "-1"
  The value of the variable "$ConstTest$" is now: "-1"

Set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
  The value of the variable "$CompValue$" is now: "-1"

If
  $ConstTest$ = $CompValue$ <<< result true
  ($ConstTest$ = $CompValue$) <<< result true
```

```

Then
  comment: passed
  comment: 1.2.13.4.5 is lower then 1.2.3.4.5
  comment: using CompareDotSeparatedStrings give wrong results on numbers

Else
EndIf

Set $string1$ = "1.2.3.4.5"
  The value of the variable "$string1$" is now: "1.2.3.4.5"

Set $string2$ = "1.2.13.4.5"
  The value of the variable "$string2$" is now: "1.2.13.4.5"

Set $ConstTest$ = "1"
  The value of the variable "$ConstTest$" is now: "1"

Set $CompValue$ = CompareDotSeparatedStrings($string1$, $string2$)
  The value of the variable "$CompValue$" is now: "1"

If
  $ConstTest$ = $CompValue$ <<< result true
  ($ConstTest$ = $CompValue$) <<< result true
Then
  comment: passed
  comment: 1.2.3.4.5 is higher then 1.2.13.4.5
  comment: using CompareDotSeparatedStrings give wrong results on numbers

Else
EndIf

```

### 7.3.13 (String-) Functions for Licence Management

- `DemandLicenseKey(`poolId [, productId [,windowsSoftwareId]])``  
 asks the opsi service via the function `getAndAssignSoftwareLicenseKey` for a reservation of a licence for the client. The pool from which the licences is taken may be explicitly given by its ID or is identified via an associated product ID or Windows Software Id (possible, if these associations are defined in the licences configuration).  
*poolId*, *productId*, *windowsSoftwareId* are Strings (resp. String expressions).  
 If no *poolId* is explicitly given, the first parameter has to be an empty String "". The same procedure is done with other not explicit given Ids.  
 The function returns the licence key that is taken from the pool.

Examples:

```

set $mykey$ = DemandLicenseKey ("pool_office2007")
set $mykey$ = DemandLicenseKey ("", "office2007")
set $mykey$ = DemandLicenseKey ("", "", "{3248F0A8-6813-11D6-A77B}")

```

- `FreeLicense(`poolId [, productId [,windowsSoftwareId]])``  
 asks the opsi service via the function `freeSoftwareLicense` to release the current licence reservation. The syntax is analogous to the syntax for `DemandLicenseKey`

Example:

```
DefVar $opsiresult$
set $opsiresult$ = FreeLicense("pool_office2007")
```

*\$opsiresult\$* becomes the empty String, if no error occurred, and, if an error occurred, the error info text.

### 7.3.14 Retrieving Error Infos from Service Calls

- `getLastServiceErrorClass`  
returns, as its name says, the class name of the error information of the last service call. If the last service call did not produce an error the function returns the value "None".
- `getLastServiceErrorMessage`  
returns the message String of the last error information resp. "None".  
Since the message String is more likely to be changed, it is recommended to base script logic on the class name.

Example:

```
if getLastServiceErrorClass = "None"
    comment "kein Fehler aufgetreten"
endif
```

## 7.4 String List Functions and String List Processing

A String list (or a String list value) is a sequence of String values. For this kind of values we have the variable of type String list. They are defined by the statement

```
DefStringList <VarName>
```

A String list value may be assigned to String list variable:

```
Set <VarName> = <StringListValue>
```

String list values can be given only as results of String expressions. There are many ways to create or capture String lists, and many options for processing them, often yielding new String lists. They are presented in the following subsections.

For the following examples we declare a String list variable *\$list1\$*:

```
DefStringList $list1$
```

If we refer to variables named like String0, StringVal, .. it is meant that these represent any String expressions.

We start with a special and rather useful kind of String lists: *maps* – also called hashes or associative arrays – which consist of a lines of the form *KEY=VALUE*. In fact, each map should establish a function which associates a *VALUE* to a *KEY*, and any *KEY* should occur at most once as the first part of a line (whereas different *KEY*'s *may be associated with identical 'VALUE* parts).

### 7.4.1 Info Maps

- `getMSVersionMap`  
get OS informations and wrties them to hash map string list.  
There are the folowinging keys:
- `major_version`
- `minor_version`

- build\_number
- platform\_id
- csd\_version
- service\_pack\_major
- service\_pack\_minor
- suite\_mask
- product\_type\_nr
- 2003r2

The Results from *suite\_mask* and *product\_type\_nr* are integers that can be build by *or* operations of the following values.

product\_type\_nr

```
0x00000001 (VER_NT_WORKSTATION)
0x00000002 (VER_NT_DOMAIN_CONTROLLER)
0x00000003 (VER_NT_SERVER)
```

SuiteMask

```
0x00000001 (VER_SUITE_SMALLBUSINESS)
0x00000002 (VER_SUITE_ENTERPRISE)
0x00000004 (VER_SUITE_BACKOFFICE)
0x00000008 (VER_SUITE_COMMUNICATIONS)
0x00000010 (VER_SUITE_TERMINAL)
0x00000020 (VER_SUITE_SMALLBUSINESS_RESTRICTED)
0x00000040 (VER_SUITE_EMBEDDEDNT)
0x00000080 (VER_SUITE_DATACENTER)
0x00000100 (VER_SUITE_SINGLEUSERTS)
0x00000200 (VER_SUITE_PERSONAL)
0x00000400 (VER_SUITE_SERVERAPPLIANCE)
```

Example:

The Code

```
DefStringList $INST_Resultlist$
DefStringList $INST_Resultlist2$

message "getMSVersionMap"
comment "get value by winst function"
set $INST_Resultlist$ = getMSVersionMap
```

produces the following log:

```
message getMSVersionMap
comment: get value by winst function

Set $INST_Resultlist$ = getMSVersionMap
  retrieving strings from getMSVersionMap [switch to loglevel 7 for debugging]
  (string 0)major_version=5
  (string 1)minor_version=1
  (string 2)build_number=2600
  (string 3)platform_id=2
```

```
(string 4)csd_version=Service Pack 3
(string 5)service_pack_major=3
(string 6)service_pack_minor=0
(string 7)suite_mask=256
(string 8)product_type_nr=1
(string 9)2003r2=false
```

**Note**

Background infos for getMSVersionMap

- <http://msdn.microsoft.com/en-us/library/ms724385%28VS.85%29.aspx>
- <http://msdn.microsoft.com/en-us/library/dd419805.aspx>
- <http://msdn.microsoft.com/en-us/library/ms724833%28VS.85%29.aspx>

- `getFileInfoMap(<FILENAME>)`  
retrieves the version infos built into the file FILENAME and writes it to a Stringlist map.

At this moment, there exist the keys,

- Comments
- CompanyName
- FileDescription
- FileVersion
- InternalName
- LegalCopyright
- LegalTrademarks
- OriginalFilename
- PrivateBuild
- ProductName
- ProductVersion
- SpecialBuild
- Language name <index>
- Language ID <index>
- file version with dots
- file version
- product version

Usage: If we define and call

```
DefStringList FileInfo
DefVar $InterestingFile$
Set $InterestingFile$ = "c:\program files\my program.exe"
set FileInfo = getFileInfoMap($InterestingFile$)
```

we get the value associated with key "FileVersion" from the call

```
DefVar $result$
set $result$ = getValue("FileVersion", FileInfo)
```

(for the function `getValue` cf. Section 7.4.4).

Example:

The code:

```
set $InterestingFile$ = "%windir%\winst.exe"
if not (FileExists($InterestingFile$))
    set $InterestingFile$ = "%windir%\winst32.exe"
endif
set $INST_Resultlist$ = getFileInfoMap($InterestingFile$)
```

produce the log:

```
Set $InterestingFile$ = "N:\develop\delphi\winst32\trunk\winst.exe"
The value of the variable is now: "N:\develop\delphi\winst32\trunk\winst.exe"

If
    Starting query if file exist ...
    FileExists($InterestingFile$) <<< result true
    not (FileExists($InterestingFile$)) <<< result false
Then
EndIf

Set $INST_Resultlist$ = getFileInfoMap($InterestingFile$)
retrieving strings from getFileInfoMap [switch to loglevel 7 for debugging]
(string 0)Language name 0=Deutsch (Deutschland)
(string 1)Language ID 0=1031
(string 2)file version=1125942857039872
(string 3)file version with dots=4.10.8.0
(string 4)product version=1125942857039872
(string 5)Comments=
(string 6)CompanyName=uib gmbh (www.uib.de)
(string 7)FileDescription=opsi.org
(string 8)FileVersion=4.10.8.0
(string 9)InternalName=
(string 10)LegalCopyright=uib gmbh under GPL
(string 11)LegalTrademarks=opsi
(string 12)OriginalFilename=
(string 13)PrivateBuild=
(string 14)ProductName=opsi-winst
(string 15)ProductVersion=4.0
(string 16)SpecialBuild=
```

- `getLocaleInfoMap`  
retrieves the system informations on the locale and writes it to a Stringlist map.

At this moment, there exist the keys:

- `language_id_2chars` (two-letter version of the system default language name)
- `language_id` (three-letter version of it, including subtype of language) inklusive der Sprachenuntertypen)
- `localized_name_of_language`

- English\_name\_of\_language
- abbreviated\_language\_name
- native\_name\_of\_language
- country\_code
- localized\_name\_of\_country
- English\_name\_of\_country
- abbreviated\_country\_name
- native\_name\_of\_country
- default\_language\_id
- default\_language\_id\_decimal
- default\_country\_code
- default\_oem\_code\_page
- default\_ansi\_code\_page
- default\_mac\_code\_page
- system\_default\_language\_id Hexadecimal Windows locale Id
- system\_default\_posix Language\_Region (Posix Style)
- system\_default\_lang\_region Language-Region (BCP 47 Style)

The `system_default` keys gives information about the language of the installed OS. The other keys give information about the locale of the GUI.

Example:

The code:

```
message "Locale Infos"
set $INST_Resultlist$ = getLocaleInfoMap
```

produces e.g the log:

```
message Locale Infos

Set $INST_Resultlist$ = getLocaleInfoMap
retrieving strings from getLocaleInfoMap [switch to loglevel 7 for debugging]
(string 0)language_id_2chars=DE
(string 1)language_id=DEU
(string 2)localized_name_of_language=Deutsch (Deutschland)
(string 3)English_name_of_language=German
(string 4)abbreviated_language_name=DEU
(string 5)native_name_of_language=Deutsch
(string 6)country_code=49
(string 7)localized_name_of_country=Deutschland
(string 8)English_name_of_country=Germany
(string 9)abbreviated_country_name=DEU
(string 10)native_name_of_country=Deutschland
(string 11)default_language_id=0407
(string 12)default_language_id_decimal=1031
(string 13)default_country_code=49
```

```
(string 14)default_oem_code_page=850
(string 15)default_ansi_code_page=1252
(string 16)default_mac_code_page=10000
(string 17)system_default_language_id=0407
(string 18)system_default_posix=de_DE
(string 19)system_default_lang_region=de-DE
```

Usage: If we define and call

```
DefStringList $languageInfo$
set $languageInfo$ = getLocaleInfoMap
```

we get the value associated with key "language\_id\_2chars" from the call

```
DefVar $result$
set $result$ = getValue("language_id_2chars", $languageInfo$)
```

(for the function `getValue` cf. Section 7.4.4). We may now write scripts using a construct like

```
if getValue("language_id_2chars", languageInfo) = "DE"
    ; installiere deutsche Version
else
    if getValue("language_id_2chars", languageInfo) = "EN"
        ; installiere englische Version
    endif
endif
```

---

### Note

Background infos for `getLocaleInfoMap`:

- <http://msdn.microsoft.com/en-us/library/cc233968.aspx>
  - <http://msdn.microsoft.com/en-us/library/0h88fahh.aspx>
  - bcp 47 validator:  
<http://schneegans.de/lv/?tags=de-de-1996&format=text>
  - <http://www.iana.org/assignments/language-subtag-registry>
  - <http://www.the-localization-tool.com/?p=698>
- 

- `getLocaleInfo`  
(deprecated): use `GetLocaleInfoMap` .
- `getProductMap` // since 4.11.2.1  
returns a info map of the opsi product you are just installing.  
It works only if *opsi-winst* is running in opsi service mode.  
keys are: id, name, description, advice, productversion, packageversion, priority, installationstate, lastactionrequest, lastactionresult, installedversion, installedpackage, installedmodificationtime

Example:

```
set $INST_Resultlist$ = getProductMap
set $string1$ = getValue("id", $INST_Resultlist$)
```

produces e.g the log:

```

Set $INST_Resultlist$ = getProductMap
retrieving strings from getProductMap [switch to loglevel 7 for debugging]
(string 0)id=opsi-winst-test
(string 1)name=opsi-winst test
(string 2)description=Test and example script for opsi-winst
(string 3)advice=
(string 4)productversion=4.11.2
(string 5)packageversion=1
(string 6)priority=0
(string 7)installationstate=unknown
(string 8)lastactionrequest=setup
(string 9)lastactionresult=successful
(string 10)installedversion=4.11.2
(string 11)installedpackage=1
(string 12)installedmodificationtime=

Set $string1$ = getValue("id", $INST_Resultlist$)
retrieving strings from $INST_Resultlist$ [switch to loglevel 7 for debugging]
(string 0)id=opsi-winst-test
(string 1)name=opsi-winst test
(string 2)description=Test and example script for opsi-winst
(string 3)advice=
(string 4)productversion=4.11.2
(string 5)packageversion=1
(string 6)priority=0
(string 7)installationstate=unknown
(string 8)lastactionrequest=setup
(string 9)lastactionresult=successful
(string 10)installedversion=4.11.2
(string 11)installedpackage=1
(string 12)installedmodificationtime=

The value of the variable "$string1$" is now: "opsi-winst-test"

```

## 7.4.2 Producing String Lists from Strings

- `createStringList (<String0, String1 ,... >`)`  
forms a String list from the values of the listed String expressions. For example, by

```
set $list1$ = createStringList ('a','b', 'c', 'd')
```

we get a list of the first four letters of the alphabet.

The following two functions produce a String list by splitting some string:

- `splitString (<string1>, <string2>)`  
generates the list of partial strings of <string1> (including empty strings) before resp. between the occurrences of <string2>. E.g.,

```
set $list1$ = splitString ("\\server\share\directory", "\\")
```

defines the list

```
"", "", "server", "share", "directory"
```

- `splitStringOnWhiteSpace (<string>)`  
slices StringVal by the "white spots" in it. E. g.

```
set $list1$ = splitStringOnWhiteSpace("Status   Lokal   Remote   Netzwerk")
```

produces the list

```
"Status", "Lokal", "Remote", "Netzwerk"
```

no matter how many blanks or tabs constitute the white space between the words.

### 7.4.3 Loading Lines of a Text File into a String List

- `loadTextFile (<file name>)`  
reads the file <file name> and generates the string list, that contains all lines of the file.
- `loadUnicodeTextFile (<file name>)`  
reads the unicode text file <file name> and generates the string list, that contains all lines of the file.  
By this call, the strings are converted into the system default 8 bit code.
- `getSectionNames (<file name>)`  
interprets the specified file as an inifile, looks for list of all lines of form  
[<SectionName>]  
and returns the pure section names (without brackets).

### 7.4.4 Simple String Values generated from String Lists

- `composeString (<string list>, <link string>)`  
With this function, the elements of any String list can be glued to one another, mediated by a "glue string".  
E.g. if `$list1$` represents the list `a, b, c, d, e`  
by

```
$line$ = composeString ($list1$, " | ")
```

we assign the value `"a | b | c | d | e"`. to `$line$`.

- `takeString(<index>,<list>)`

A. g., if `$list1$` represents the list of the first five letters of the alphabet by

```
takeString (2, $list1$)
```

we get string "c" (since list counting starts with 0).

Negative values of index go downwards from the list count value. E.g.,

```
takeString (-1, $list1$)
```

return the last list element, that is "e".

- `takeFirstStringContaining(<list>,<search string>)`  
returns the first string of the list which contains the <search string>.  
Returns an empty string if no matching string was found.

- `getValue (<key>, <list>)`  
This function tries to interpret a String list as list of lines of the form *key=value*. It looks for the first line, where the string <key> is followed by the equality sign, and returns the remainder of the line (the *value*, the string that starts after the equality sign). If there is no fitting line, it returns the string *NULL*. The function is required for using the `getLocaleInfoMap` and `getFileVersionMap` string list functions (cf. Section 7.4.1).
- `getValueBySeparator(<key string>,<separator string>,<hash string list> )` //since 4.11.2.1  
works like `getValue` but you have to give the <separator string> so that can also work with hashes like *key:value*
- `count (<list>)`  
returns the number of elements of the string list <list> as string.  
e.g. for `$list1$` composed as  
*a, b, c, d, e*  
`count ($list1$)` has the value "5".

### 7.4.5 Producing String Lists from opsi-winst Sections

- `retrieveSection (`section name`)`  
gives the lines of the specified section as string list.
- `getOutputStreamFromSection (`section name`)`  
invokes the section and – at this moment implemented only for `DosInAnIcon` (`ShellInAnIcon`), `ExecWith` and `ExecPython` calls – captures the output to standard out and standard error of the invoked commands writing them into a string list. For example:

```
set $list$ = getOutputStreamFromSection ('DosInAnIcon_netuse')

[DosInAnIcon_netuse]
net use
```

`$list1$` contains among some surrounding stuff the list of all mounted shares of a PC.

- `getReturnListFromSection (`section name`)`  
For some section types - at this moment implemented only for `XMLPatch` sections and `opsiServiceCall` sections - there is a specific `return` statement which yields some result of the execution of the section (assumed to be of String list type).  
E.g. we may use the statement

```
set list1 = getReturnListFromSection ('XMLPatch_mime "c:\mimetypes.rdf"')
```

to get a specific knot list of the XML file `mimetypes.rdf`. (More info to `XMLPatch` sections at Section 8.7 in this manual).

Or the list of opsi clients is produced by the reference to a opsi service call:

```
DefStringList $result$
Set $result$=getReturnListFromSection("opsiservicecall_clientIdsList")

[opsiservicecall_clientIdsList]
"method": "getClientIds_list"
"params": []
```

## 7.4.6 Transforming String Lists

- `getSubList (<Startindex>, <Endindex>, <list> )`  
returns a partial list of a given list.  
E.g., if list represents the list of letters *a, b, c, d, e*, by the statement:

```
set $list1$ = getSubList(1 : 3, $list$)
```

we get the partial list *b, c, d*. Begin index as well as end index have to be interpreted as the index of the first and last included list elements. The counting starts with 0.

Default start index is 0, default end index is the index of the last element of the list.

Therefore, (for the above defined list1) the command

```
set $list1$ = getSubList(1 : , $list$)
```

yields the list *b, c, d, e*.

```
set $list1$ = getSubList(:, $list$)
```

produces a copy of the original list.

It is possible to count backwards in order to determine the last index:

```
set $list1$ = getSubList(1 : -1, $list$)
```

defines the list of elements starting with the first and ending with the second to last element of the list – in the above example we again get list *b, c, d*.

- `getListContaining(<list>,<search string>)`  
returns the first string from <list> which contains <search string>. Returns empty string if <search string> is not found.
- `takeFirstStringContaining(<list>,<search string>)`  
Liefert den ersten String von <list> welcher den <search string> enthält. + Liefert einen Leerstring wenn <search string> nicht gefunden wird.
- `addtolist(<list>,<string>)`  
Appends <string> to the list <list>.
- `addlisttolist(<list1>,<list2>)`  
Appends the list <list2> to the list <list1>.
- `reverse (<list>)`  
produces the inverted list,  
if \$list\$ is *a, b, c, d, e*, by

```
set $list1$ = reverse ($list$)
```

we get the \$list1\$ *e, d, c, b, a*.

## 7.4.7 Iterating through String Lists

An important usage of string lists is based on the possibility that the script runs through all elements of a list executing some operation on each string element.

The syntax to define this repetition is:

- `for %s% in <list> do <one statement | sub section>`

This expression locally defines a string variable `%s%` that takes one by one the values of the list elements. `<one statement>` can be any single statement that can exist in a primary section or (and most interestingly) it may be a subsection call. The locally defined iteration index `%s%` exists in the whole context of statement, in particular in the subsection if statement is a subsection call.



#### Caution

The replacement mechanism for `%s%` always works like that for constants: The name of the variable is replaced by the element values. If we iterate through a list `a,b,c` and the iteration index is named `%s%`, we get for `%s%` one by one `a`, `b`, `c` – not the String values. To reproduce the original list elements we have to enclose `%s%` in citation marks.

Example: Let `$list1$` be the list `a, b, c, d, e`, and `$line$` a String variable. The statement

```
for %s% in $list1$ do set $line$ = $line$ + "%s%"
```

iterates through the list elements internally executing

```
$line$ = $line$ + "a"
$line$ = $line$ + "b"
$line$ = $line$ + "c"
$line$ = $line$ + "d"
$line$ = $line$ + "e"
```

Such, finally line has value `abcde`. If we omitted the citation marks around `%s%` we would get a syntax error for each iteration step.

Please note: The note variable is only valid in the directly called procedure. If it is needed in sub programs of it its value must be transferred to a global variable.

## 7.5 Special Commands

- `Killtask <process>` tries to stop all processes that execute the program named by the string expression.  
E.g.

```
killtask "winword.exe"
```

- `sleepSeconds <Integer>`  
breaks the program execution for `<Integer>` seconds.
- `markTime`  
sets a time stamp for the current system time and logs it.
- `diffTime`  
logs the time passed since the last `marktime`.

## 7.6 Commands for logging control

- `comment <string>` or `comment = <const string>`  
writes the value of the String expression resp. the sequence of characters into the log file.

- `LogError <string>` or `LogError = <const string>`  
writes additional error messages to the log file and increments the error counter by one.
- `LogWarning <string>` or `LogWarning = <const string>`  
writes additional warning messages to the log file and increments the warning counter by one.
- `includeLog <file name> <tail size> //since 4.11.2.1`  
Includes the file <file name> into the log file. Only the last <tail size> lines will be included. If you start a other program (e.g. setup program) which produces a log file, you may use this command to include information from this log into the *opsi-winst* log file.  
Example:

```
includeLog "%Scriptpath%\test-files\10lines.txt" "5"
```

## 7.7 Commands for User Information and User Interaction

- `Message <string expression>`  
bzw.  
`Message = <sequence of characters>`  
lets *opsi-winst* display the value of the String expression resp. the sequence of chars in the batch window in the top information line. The text is kept as long as no new `message` is set.  
Example:

```
Message "Installation von "+$productid$
```

- `ShowMessageFile <file name>`  
interprets the String expression as text file name, tries to read the text and show it in a user information window. Execution stops until the user confirms reading. E.g. by a command like

```
ShowMessageFile "p:\login\day.msg"
```

one can realize a "Message of the Day" mechanism.

- `ShowBitMap [<image name>] [<inscription>]`  
places the image denoted by the <image name> (in BMP, JPEG or PNG format, size 160x160 pixel) and shows the inscription.  
<image name> and <inscription> are String expressions.  
Example:

```
ShowBitmap "%scriptpath%" + $ProductId$ + ".png" "$ProductId$"
```

- `Pause <string>` or `Pause = <const string>`  
display the text given as a String expression or as a sequence of chars in a information window waiting until the user confirms the continuation.
- `Stop <string>` or `stop = <const string>`  
halt program execution if the user confirms it. The String expression resp. the (possibly empty) sequence of chars explain to the user what is supposed to be stopped.

## 7.8 Commands for userLoginScripts / Roaming Profile Support

### Note

The opsi *Roaming Profile Support* is under co-funding. This means that these features are not free yet (05.10.2011)

- `GetScriptMode` //since 4.11.2.1  
give one of the possible values *Machine,Login*:
  - *Machine* - the script is **not** running as *userLoginScript*
  - *Login* - the script is running as *userLoginScript*
- `GetUserSID(<Windows Username>)`
- `GetLoggedInUser` //since 4.11.1.2
- `GetUsercontext` //since 4.11.1.2  
returns the username in whose context the *opsi-winst* is just running.
- `saveVersionToProfile` //since 4.11.2.1  
save `productversion-packageversion` to local profile  
It is designed to be used in *userLoginScripts*.  
This command is used in combination with `readVersionFromProfile` or `scriptWasExecutedBefore`. It marks that the *userLoginScript* for this product in this product version and package version was executed for the actual user. The information is saved at the file "%CurrentAppdataDir%\opsi.org\userLoginScripts.ini"
- `readVersionFromProfile` //since 4.11.2.1  
returns a string with the `productversion-packageversion` for the running opsi product which was read from local profile. See also: `saveVersionToProfile`  
It is designed to be used in *userLoginScripts*.
- `scriptWasExecutedBefore` //since 4.11.2.1  
This Boolean function `scriptWasExecutedBefore` checks if there is a version stamp in the profile (like you may do with the `readVersionFromProfile` command) It returns *true* if saved and running `productversion-packageversion` are identical. Then it set a new stamp to the profile (like you may do with the `saveVersionToProfile` command). So you may just use this single command in a *if* statement.  
It is designed to be used in *userLoginScripts*.
- `isLoginScript` //since 4.11.2.1  
This boolean function returns *true* if the script is running as *userLoginScript*. See also: `GetScriptMode`

## 7.9 Conditional Statements (if Statements)

In primary sections, the execution of a statement or a sequence of statements can be made dependent on some condition.

Example

```
;Which Windows version?
DefVar $MSVersion$

Set $MSVersion$ = GetMsVersionInfo
if ($MSVersion$ >= "6")
    sub_install_win7
else
    if ( $MSVersion$ = "5.1" )
```

```

sub_install_winXP
else
  stop "not a supported OS-Version"
endif
endif

```

### 7.9.1 General Syntax

The syntax of the complete if statement is:

```

if <condition>
<sequence of statements>
else
<sequence of statements>
endif

```

The else part may be omitted.

if statements may be nested. That is, in the sequence of statements that depend on an if clause (no matter if inside the if or the else part) another if statement may occur.

<condition> is a <Boolean expression> . A Boolean (or logical) expression can be constructed as a (String) value comparison, by Boolean operators, or by certain function calls which evaluate to true or false. Up to now these Boolean values cannot be explicitly represented in a *opsi-winst* script).

### 7.9.2 Boolean Expressions

The String comparison (which is a Boolean expression) has the form

```
<String expression> <comparison sign> <String expression>
```

where <comparison sign> is one of the signs

```
< <= >= >
```

String comparisons in *opsi-winst* are case independent.

Inequality must be expressed by a NOT() expression which is presented below.

There is as well a comparison expression for comparing Strings as (integer) numbers. If any of them cannot be converted to a number an error will be indicated.

This number comparison expression has the same form as the String comparison but for an INT prefix of the comparison sign:

```
<String expression> INT<comparison sign> <String expression>
```

Such, we can build expressions as

```
if $Name1$ INT<= $Name2$
```

or

```
if $Number1$ INT>= $Number2$
```

Boolean operators are AND, OR, and NOT() (case does not matter). If b1, b2 and b3 are Boolean expressions the combined expressions

```
b1 AND b2
```

```
b1 OR b2
```

```
NOT( b3 )
```

are Boolean expressions as well denoting respectively the conjunction (AND), the disjunction (OR) and the negation (NOT).

A Boolean expression can be enclosed in parentheses (such producing a new Boolean expression with the same value).

The common rules of Boolean operator priority ("and" before "or") are at this moment not implemented. An expression with more than one operator is interpreted from left to right. For clarity, in a Boolean expression that combines AND and OR operators parentheses should be employed, e.g. we should explicitly write `b1 OR (b2 AND b3)`

or

`(b1 OR b2) AND b3`

The second example describes what would be executed if there were no parentheses - whereas the common interpretation would run as the other line indicates.

Boolean operators can be conceived as special Boolean valued functions (the negation operator demonstrates this very clearly).

There are some more Boolean functions implemented. Every call of such a function constitutes a Boolean expression as well:

- `FileExists(<file name>)`  
returns *true* if the denoted file or directory exists otherwise *false*.
- `FileExists32(<file name>)` see [Chapter 64 Bit support](#)
- `FileExists64(<file name>)` see [Chapter 64 Bit support](#)
- `FileExistsSysNative(<file name>)` see [Chapter 64 Bit support](#)
- `LineExistsIn(<line>, <file name>)`  
returns *true* if the text file denoted by <file name> contains a line as specified in the first parameter where each parameter is a String expression. Otherwise (or if the file does not exist) it returns *false*.
- `LineBeginning_ExistsIn(<string>, <file name>)`  
returns *true* if there is line that begins with <string> in the text file denoted by filename (each parameter being a string expression). Otherwise (or if the file does not exist) it returns *false*.
- `XMLAddNamespace(<XMLfilename>, <XMLelementname>, <XMLnamespace>)`  
inserts a XML namespace definition into the first XML element with the given name (if not existing). It gives back if an insertion took place. (The *opsi-winst* XML patch section need the definitions of namespace.)  
The file must be formatted that an element tag has no line breaks in it. For an example, cf. [cookbook Section 10.6](#).
- `XMLRemoveNamespace(<XMLfilename>, <XMLelementname>, <XMLnamespace>)`  
removes the XML namespace definition from the XML element. It gives back if an removal took place. We need this to simulate that an original file is unchanged. For an example, cf. [cookbook Section 10.6](#).
- `HasMinimumSpace(<Laufwerksname>, <Kapazität>)`  
returns true if at least a capacity capacity is left on drive drivename. capacity as well as drivename syntactically are String expressions. The capacity may be given as a number without unit specification (then interpreted as bytes) or with unit specifications "kB", "MB", or "GB" (case independent).  
Example:

```
if not (HasMinimumSpace ("%SYSTEMDRIVE%", "500 MB"))
  LogError "Not enough space on %SystemDrive%, 500MB on drive %SystemDrive% needed"
  isFatalError
endif
```

- `opsiLicenseManagementEnabled`  
returns *true* if the opsi license management module is enabled.
- `runningAsAdmin`  
returns *true* if the script is running in an administrative account.

## 7.10 Subprogram Calls

Statements in primary sections which refer to instructions declared elsewhere are subprogram calls.,

```
if ($MSVersion$>="6")
    sub_install_win7
else
    if ( $MSVersion$ = "5.1" )
        sub_install_winXP
    else
        stop "not a supported OS-Version"
    endif
endif
```

In this example the statement:

```
sub_install_winXP
```

"calls" the section titled *[sub\_install\_winXP]* which is placed somewhere else in the script. E.g. we may have

```
[sub_install_winXP]
Files_copy_XP
WinBatch_SetupXP
```

Generally, there are three ways to place the referred instructions:

- The most common target of a sub program call is some other internal section in the very script file where the calling statement is placed (as in the example).
- We may put the referred instructions into another file which serves as an external section.
- Any String list can be used as list of instructions for a sub program call.

We describe the syntax of sub program calls in detail:

### 7.10.1 Syntax of Procedure Calling

Formally, the syntax can be given by

```
<proc. type>( <proc. name> | <External proc. file> | <String list function> )
```

This expression may supplemented by one or more parameters (procedure type dependent).

That means: A procedure call consists of three main parts.

The first part is the subprogram type specifier.

Examples of type names are *Sub* (we call a procedure of type sub that is again a primary section) or *Files* and *WinBatch* (calls of special secondary sections). The complete overview of the existing sub program types is given at Section 7.10.

The second part determines where and how the lines of sub program are to be found.

1. The subprogram is a sequence of lines situated in the executed *opsi-winst* script as another internal section. Then a name (constituted from letters, digits, and some special characters) has to be appended to the type specifier (without space) in order to form an unique section name.

```
sub_install_winXP
or
files_copy_winXP
```

Section names are case independent as any other string.

2. If the type specifier stands alone a String list expression or a String expression is expected. If the expression following the type specifier cannot be resolved as a String list expression (cf. case (3)) it is assumed to be a String expression. The string is then interpreted as a file name. *opsi-winst* tries to open the file as a text file and interprets its lines as an external section of the specified type.  
E.g.  
sub "p:\install\opsiutils\mainroutine.ins" tries to execute the lines of mainroutine.ins as statements of a sub section.
3. If the expression following a pure section type specifier is resolvable as a String list expression the lines of the list are interpreted as the statements of the section.  
This mechanism can e.g. be used to load a file that has unicode format and then treat it by the usual mechanisms

```
registry loadUnicodeTextFile("%scriptpath%/opsiorgkey.reg") /regedit
```

Syntactically, this line is composed of three main parts:

- \* **registry**, the core statement specifying the section type,
- \* **loadUnicodeTextFile(...)**, a String list expression specifying how to get the lines of a registry section resp. its surrogate.
- \* **/regedit**, parametrizing the registry call.

In this example, the call parameter already gives an example for the third part of a subsection call:

The third part of a procedure call comprises type specific call options.

For a reference of the call options cf. the descriptions of the section calls in Chapter 8.

## 7.11 Controlling Reboot

The command **ExitWindows** is used to control reboots, shutdown and similar actions which should take place after the *opsi-winst* it self is terminated. The name of the command and the fact that there is no *ExitWindows* without modifier has historical reasons: Working on Windows 3.1 you could exit windows to go back to the DOS level.

- **ExitWindows /RebootWanted**  
DEPRECATED: a reboot request is registered which should be executed when all installations requests are treated, and the last script has finished.  
In fact, this command is now treated as an **ExitWindows /Reboot** (since otherwise an installation could fail because a required product is not yet completely installed).
- **ExitWindows /Reboot**  
triggers the reboot after *opsi-winst* has finished the currently treated script.
- **ExitWindows /ImmediateReboot**  
breaks the normal execution of a script anywhere inside it. When this command is called *opsi-winst* runs as directly as possible to its end entailing the system **ExitWindows** call. In the context of an installed opsi-client-agent it is guaranteed that after rebooting *opsi-winst* runs again into the script that was aborted. Therefore, the script has to take provisions that the execution continues after the point where it was left the turn before (otherwise we may get an infinite loop ...) Cf. the example in this section.
- **ExitWindows /ImmediateLogout**  
The normal execution of a script breaks at the point of the call, and the *opsi-winst* stops running. This behaviour is needed if an automated user log in for some other user shall take place (cf. Section 10.3).
- **ExitWindows /ShutdownWanted**  
sets a flag in the registry that the PC shuts down when all installations requests are treated, and the last script has finished.

How flags may be set to ensure that the script does not run into an infinite loop when **ExitWindows /ImmediateReboot** is called we demonstrate by the following code fragment:

```

DefVar $OS$
DefVar $Flag$
DefVar $WinstRegKey$
DefVar $RebootRegVar$

set $OS$=EnvVar("OS")

if $OS$="Windows_NT"

Set $WinstRegKey$ = "HKLM\SOFTWARE\opsi.org\winst"
Set $Flag$ = GetRegistryStringValue(["+$WinstRegKey$+" "RebootFlag")

if not ($Flag$ = "1")
;=====
; Statements BEFORE Reboot

Files_doSomething

; initialize reboot ...
Set $Flag$ = "1"
Registry_SaveRebootFlag
ExitWindows /ImmediateReboot

else
;=====
; Statements AFTER Reboot

; set back reboot flag
Set $Flag$ = "0"
Registry_SaveRebootFlag

; the work part after reboot:

Files_doMore

endif

endif

[Registry_SaveRebootFlag]
openKey [$WinstRegKey$]
set "RebootFlag" = "$Flag$"

[Files_doSomething]
; a section executed before reboot

[Files_doMore]
; a section executed after reboot

```

## 7.12 Keeping Track of Failed Installations

If a product installation fails this should be signaled to the server.

According to the fact that there is no automatism to detect a failed installation this have to be done by script commands.

To indicate in a *opsi-winst* script that the installation is failed we have to call the statement:

```
isFatalError
```

If this statement is called *opsi-winst* stops the normal execution of the script and sets the product result to *failed* (otherwise it is *success*).

E. g. , a "fatal error" shall be triggered if there is as much space left as it is needed for an installation:

```
DefVar $SpaceNeeded$
Set $SpaceNeeded$ = "200 MB"

DefVar $LogErrorMessage$
Set $LogErrorMessage$ = "Not enough space on drive . Required "
Set $LogErrorMessage$ = $LogErrorMessage$ + $SpaceNeeded$"

if not(HasMinimumSpace ("%SYSTEMDRIVE%", $SpaceNeeded$))
  LogError $LogErrorMessage$
  isFatalError
  ; finish execution and set ProductState to failed

else
  ; we start the installation
  ; ...

endif
```

It is also possible to state

```
isFatalError
```

depending on the number of errors which occurred in some critical part of an installation script. In order to do this we initialize the error counting by the command

```
* markErrorNumber
```

The number of execution errors which occur after setting the counter can be queried by the the number valued function

```
* errorsOccuredSinceMark
```

We can evaluate the result in a numerical comparison condition (that as yet is only implemented for this expression).

E. g. we may state

```
* if errorsOccuredSinceMark > 0
```

and may, if this seems to make sense, then state

```
isFatalError
```

For increasing the number of counted errors depending on certain circumstances (that do not directly produce an error) we may use the `logError` statement.

We may test this by the following script example:

```
markErrorNumber
; Errors occurring after this mark are counted and
; will possibly be regarded as fatal

logError "test error"
; we write "test error" into the log file
; and increase the number of errors by 1
; for testing, comment out this line

if errorsOccuredSinceMark > 0
  ; we finish script execution as quick as possible
  ; and set the product state to "failed"

isFatalError
```

```
    ; but comment writing is not stopped  
    comment "error occured"  
else  
    ; no error occured, lets log this:  
    comment "no error occured"  
endif
```

## Chapter 8

# Secondary Sections

The secondary sections can be called from any primary section but have a different syntax. The syntax is derived from the functional requirements and library conditions and conventions for the specific purposes. Therefore from a secondary section, no further section can be called.

Secondary sections are specific each for a certain functional area. This refers to the object of the functionality, e.g. file system in general, the Windows registry, or XML files. But it refers even more to the apparatus that is internally applied. This may be demonstrated by the the variants of the batch sections (which call external programs or scripts).

The functional context is mirrored in the specific syntax of the particular section type.

### 8.1 Files Sections

A Files section mainly offers functions which correspond to copy commands of the underlying operating system. The surplus value when using the *opsi-winst* commands is the detailed logging and checking of all operations when necessary. If wanted overwriting of files can be forbidden if newer versions of a file (e.g. an newer dll-file) are already installed on the system.

#### 8.1.1 Example

A simple Files section could be:

```
[Files_do_some_copying]
copy -sV "p:\install\instnsc\netscape\*.*" "C:\netscape"
copy -sV "p:\install\instnsc\windows\*.*" "%SYSTEMROOT%"
```

These commands cause that all files of the directory `p:\install\instnsc\netscape` are copied to the directory `C:\netscape`, and then all files from `p:\install\instnsc\windows` to the windows system directory (its value is automatically inserted into the constant name `%SYSTEMROOT%`). Option `-s` means that all subdirectories are copied as well, `-V` activates the version control for library files.

#### 8.1.2 Modifier

In most cases a Files section will be called without parameters.

There are only some special uses of Files sections where the target of copy actions is set or changed in a certain specified way. We have got the two optional parameters

- `/AllNTUserProfiles` resp.

- /AllNTUserSendTo

Both variants mean:

The called Files section is executed once for each local Windows NT user. Every copy command in the section is associated with an user specific target directory.

In case other we need to build other user specific path names we can use the automatically set variable %UserProfileDir% or since *opsi-winst* version 4.11.2 %CurrentProfileDir%. With option /AllNTUserProfiles the user specific target directory for copy actions is the user profile directory (that is usually denoted by the user name and is by default situated as a subdirectory of the userappdata directory. In case of option /AllNTUserSendTo the target directory is the path of the user specific *SendTo* folder (for links of the windows explorer context menu).

The exact rule for determining the target path for a copy command has three parts:

1. If only the source of a copy action is specified the files are copied directly into the user target directory. We have syntax  
`copy <source file(s)>`  
 It be equivalent as  
`copy <source file(s)> "%UserProfileDir%"`  
 or since 4.11.2  
`copy <source file(s)> "%CurrentProfileDir%"`
2. If some targetdir is specified and targetdir is a relative path description (starting neither with a drive name nor a backslash) then targetdir is regard as the name of a subdirectory of the user specific directory. I.e.  
`copy <source file(s)> <targetdir>`  
 is interpreted like:  
`copy <source file(s)> "%UserProfileDir%\targetdir"`  
 or since 4.11.2  
`copy <source file(s)> "%CurrentProfileDir%\targetdir"`

The use of %CurrentProfileDir% has the advantage that you may the same *Files* section with /AllNTUserProfiles if it is not running as *userLoginScript* (in *Machine* script mode) and without /AllNTUserProfiles if it is running as *userLoginScript* (in *Login* script mode).

1. If targetdir is an absolute path it is used as the static target path of the copy action.

There are also the Options:

- /32Bit
- /64Bit
- /SysNative

which manipulate the *file redirection* on 64 Bit systems. For more details see Chapter 9

### 8.1.3 Commands

In a Files section the following commands are defined:

- Copy
- Delete
- del
- SourcePath
- CheckTargetPath

**Copy** and **Delete** roughly correspond to the Windows shell commands `xcopy` resp. `del`.

**SourcePath** and **CheckTargetPath** set origin and destination of the forthcoming copy actions (as if we would open two explorer windows for copy actions between them). If the target path does not exist it will be created.

The syntax definitions are:

- **Copy** [-svdunxwnr] <source(mask)> <target path>

The source files can be denoted explicitly, using the wild card sign ("\* ") or by a directory name.



### Caution

The <target path> is always understood as a directory name. Renaming by copying is not possible. If the target path does not exist it will be created (if needed a hierarchy of directories).

The optional modifiers of the **Copy** command mean (the ordering is insignificant):

- **s** → We recursive into subdirectories.
- **e** → Empty Subdirectories.  
If there are empty subdirectories in the source path they will be created in the target directory as well.
- **V** → Version checking  
A newer version of a windows library file is not overwritten by an older one (according primarily to the internal version counting of the file). If there are any doubts regarding the priority of the files a warning is added to the log file.
- **v** → (do not use)  
With Version checking:  
Deprecated: Don't use it on Systems higher than win2k. Because it checks not only against the target directory but also against %System%. use **-V** instead.
- **d** → With date check:  
A newer .exe file is not overwritten by an older one.
- **u** → We are only updating files:  
A file is not copied if there is a newer or equally old file of the same name.
- **x** → x-tract  
If a file is a zip archive it will be unpacked (Xtracted) on copying.  
Caution: Zip archives are not characterized by its name but by an internal definition. E.g. a java jar file is a zip file. If it is unpacked the application call will not work.
- **w** → weak  
We respect any write protection of a file such proceeding "weakly" (in opposite to the default behaviour which is to try to use administrator privileges and overwrite a write protected file).
- **n** → no over write  
Existing files are not overwritten.
- **c** → continue  
If a system file is in use, then it can be overwritten only after a reboot. The *opsi-winst* default behaviour is therefore that a file in use will be marked for overwriting after the next reboot, AND the *opsi-winst* reboot flag is set. Setting the copy modifier **"-c"** turns the automatic reboot off. Instead normal processing continues, the copying will be completed only when a reboot is otherwise triggered.
- **r** → read-only Attribute  
If a copied file has a read-only attribute it is set again (in opposite to the default behaviour which is to eliminate read-only attributes).

- **Delete** [-sfd[n]] <path>

or

- **Delete** [-sfd[n]] <source(mask)>  
deletes files and directories.

Possible options are (with arbitrary ordering)

- **s** → subdirectories  
We recurse into subdirectories. Everything that matches the path name or the source mask is deleted.

---

**Caution**  
The command  
`delete -s c:\opsi`  
Do not mean: remove the directory `c:\opsi` recursive, but it means: delete starting from `c:\` all occurrences of `opsi`. This may lead to a complete hard disk scan.  
If you want to delete the directory `c:\opsi` recursive use the command:  
`delete -s c:\opsi\`  
by using a trailing backslash you define that `opsi` is a directory.  
**It is safer to use the command `del` instead.**

---

- **f** → force  
forces to delete read only files
  - **d** [n] → date  
Only files of age n days or older are deleted. n defaults to 1.
- **del** [Options] <path[/mask] //since 4.11.2.1  
Works like `delete` but on  
`del -s -f c:\not-exists`  
if `c:\not-exists` not exists it do not search complete `c:\` for `not-exists`

Example (you may forget the trailing Backslash):

```
del -sf c:\delete_this_dir
```

- **SourcePath** = < source directory>  
Sets <source directory> as default directory for the following Copy and (!) Delete commands.
- **CheckTargetPath** = <target directory>  
Sets <target directory> as default directory for Copy command . If the specified path does not exist it will be created.

## 8.2 Patches-Sections

A Patches section modifies a property file in ini file format. I. e. a file that consists of sections which are a sequence of entries constructed as settings <variable> = <value>. where sections are characterized by headings which are bracketed names like [sectionname].

### 8.2.1 Example

```
Patches_DUMMY.INI $HomeTestFiles$+"\dummy.ini"

[Patches_dummy.ini]
add [secdummy] dummy1=add1
add [secdummy] dummy2=add2
add [secdummy] dummy3=add3
```

```

add [secdummy] dummy4=add4
add [secdummy] dummy5=add5
add [secdummy] dummy6=add6
set [secdummy] dummy2=set1
addnew [secdummy] dummy1=addnew1
change [secdummy] dummy3=change1
del [secdummy] dummy4
Replace dummy6=add6 replace1=replace1

```

produces the following log:

```

Execution of Patches_DUMMY.INI
  FILE C:\tmp\testFiles\dummy.ini
  Info: This file does not exist and will be created
addEntry [secdummy] dummy1=add1
  addSection [secdummy]
    done
  done
addEntry [secdummy] dummy2=add2
  done
addEntry [secdummy] dummy3=add3
  done
addEntry [secdummy] dummy4=add4
  done
addEntry [secdummy] dummy5=add5
  done
addEntry [secdummy] dummy6=add6
  done
setEntry [secdummy] dummy2=set1
  Entry    dummy2=add2
  changed to dummy2=set1
addNewEntry [secdummy] dummy1=addnew1
  appended entry
changeEntry [secdummy] dummy3=change1
  entry    dummy3=add3
  changed to dummy3=change1
delEntry [secdummy] dummy4
  in section secdummy deleted  dummy4=add4
replaceEntrydummy6=add6 replace1=replace1
  replaced in line 7
C:\tmp\testFiles\dummy.ini saved back

```

For more examples, please check the opsi standard product *opsi-winst-test* and in this product the part `$Flag_winst_patches$ = "on"`

## 8.2.2 Parameter

As shown in the example the name of the property file to be patched is specified as parameter of the sub program call.

## 8.2.3 Commands

For a Patches section, we have commands:

- add
- set

- **addnew**
- **change**
- **del**
- **delsec**
- **replace**

Each command refers to some section of the file which is to be patched. The name of this section is specified in brackets [] (which do here not mean "syntactically optional!!").

In detail:

- **add** [**<section name>**] **<variable1> = <value1>**  
This command adds an entry of kind **<variable1> = <value1>** to section **<section name>** if there is yet no entry for **<variable1>** in this section. Otherwise nothing is written. If the section does not exist it will be created.
- **set** [**<section name>**]**<variable1> = <value1>**  
If there is no entry for **<variable1>** in section **<section name>** the setting **<variable1> = <value1>** is added. Otherwise, the first entry **<variable1> = <valueX>** is changed to **<variable1> = <value1>**.
- **addnew** [**<section name>**]**<variable1> = <value1>**  
No matter if there is an entry for **<variable1>** in section **<section name>** the setting **<variable1> = <value1>** is added.
- **change** [**<section name>**]**<variable1> = <value1>**  
Only if there is any entry for **<variable1>** in section **<section name>** it is changed to **<variable1> = <value1>**.
- **del** [**<section name>**] **<variable1> = <value1>**  
resp.  
**del** [**<section name>**] **<variable1>**  
removes all entries **<variable1> = <value1>** resp. all entries for **<variable1>** in section **<section name>**.
- **delsec** [**<section name>**]  
removes the section **<section name>**.
- **replace** **<variable1>=<value1> <variable2>=<value2>**  
means that **<variable1> = <value1>** will be replaced by **<variable2> = <value2>** in all sections of the ini file. There must be no spaces in the value or around the equal signs.

## 8.3 PatchHosts Sections

By virtue of a PatchHosts section we are able to modify a hosts file which is to understand as any file with lines having format

*IPadress hostName aliases # comment*

*Aliases* and *comment* (and the comment separator *#*) are optional. A line may also be a comment line starting with *#*.

The file which is to be modified can be given as parameter of a *PatchHosts* call. If there is no parameter a file named HOSTS is searched in the directories *c:\nfs*, *c:\windows* and *%systemroot%\system32\drivers\etc*. If no such file is found the *PatchHosts* call terminates with an error.

In a PatchHosts section there are defined commands:

- **setAddr**
- **setName**

- `setAlias`
- `delAlias`
- `delHost`
- `setComment`

Example:

```
PatchHosts_add $HomeTestFiles$\hosts
```

```
[PatchHosts_add]
```

```
setAddr ServerNo1 111.111.111.111
setName 222.222.222.222 ServerNo2
setAlias ServerNo1 myServerNo1
setAlias 222.222.222.222 myServerNo2
setComment myServerNo2 Hallo Welt
```

produces the following log:

```
Execution of PatchHosts_add
FILE C:\tmp\testFiles\hosts
Set ipAddress 111.111.111.111 Hostname "ServerNo1"
Set Hostname "ServerNo2" for ipAddress 222.222.222.222
Alias "myServerNo1" set for entry "ServerNo1"
Alias "myServerNo2" set for entry "222.222.222"
SetComment of Host "myServerNo2" to "Hallo Welt"
C:\tmp\testFiles\hosts saved back
```

For more examples, please check the opsi standard product *opsi-winst-test* and in this product the part `$Flag_winst_patch_hosts$ = "on"`.

In detail:

- `setaddr <hostname> <ipadresse>`  
sets the IP address for host <hostname> to <IPaddress>. If there is no entry for host name as yet it will be created.
- `setname <ipadresse> <hostname>`  
sets the host name for the given IP address. If there is no entry for the IP address as yet it will be created.
- `setalias <hostname> <alias>`  
adds an alias for the host named <hostname>.
- `setalias <IPadresse> <alias>`  
adds an alias name for the host with IP address <IPaddress>.
- `delalias <hostname> <alias>`  
removes the alias name <alias> for the host named <hostname> .
- `delalias <IPadresse> <alias>`  
removes the alias name <alias> for the host with IP address <IPaddress>.
- `delhost <hostname>` removes the complete entry for the host with name <hostname>.
- `delhost <IPadresse>`  
removes the complete entry for the host with IP address <IPaddress>.
- `setComment <ident> <comment>`  
writes <comment> after the comment sign for the host with host name, IP address or alias name <ident>.

## 8.4 IdapiConfig Sections

A IdapiConfig section were designed to write parameters in idapi\*.cfg files which are used by the Borland Database Engine.

This section type is not supported any more.

## 8.5 PatchTextFile Sections

A PatchTextFile section offers a variety of options to patch arbitrary configuration files which are given as common text files (i.e. they can be treated line by line).

An essential tool for working on text files is the check if a specific line is contained in a given file. For this purpose we have got the Boolean functions `Line_ExistsIn` and `LineBeginning_ExistsIn` (cf. Section 7.9.2).

### 8.5.1 Parameter

The text file which is to be treated is given as parameter.

### 8.5.2 Commands

We have got two commands especially for patching Mozilla preferences files plus the two deprecated and more restricted older versions of these commands:

- `Set_Mozilla_Pref` ("`<preference type>`", "`<preference key>`", "`<preference value>`")  
sets for `<preference type>` the value associated with "`<preference variable>`" to "`<preference value>`".  
In current Mozilla preference files there are expressions like  
`user_pref("<key>", "<value>")`  
`pref("<key>", "<value>")`  
`lock_pref("<key>", "<value>")`  
Each of them, in fact, any (javascript) function call of the form  
`functionname (String1, String2)`  
can be patched with this command by setting the appropriate string for `<preference type>` (that is, resp. for `functionname`), If an entry starting with "`functionname (String1`" exists in the treated file, it will be patched (and left at its place). Otherwise a new line will be appended. Unusually in *opsi-winst*, all strings are case sensitive.
- `Set_Netscape_User_Pref` ("`<preference variable>`", "`<value>`")  
sets the line of the given user preference file for the variable `<preference variable>` to value `<value>`. The ASCII ordering of the file will be rebuilt.  
(Deprecated!)
- `AddStringListElement_To_Mozilla_Pref` ("`<preference type>`", "`<preference variable>`", "`<add value>`")  
appends an element to a list entry in the given preference file. It is checked if the value that should be added is already contained in the list (then it will not be added).
- `AddStringListElement_To_Netscape_User_Pref` ("`<preference variable>`", "`<add values list>`")  
(Deprecated!)

The other commands of *PatchTextFile* sections are not file type specific. All operations are based on the concept that a line pointer exists which can be moved from top of the file i.e. above the top line down to the bottom (line).

There are three search commands:

- `FindLine <search string>`

- `FindLine_StartingWith <search string>`
- `FindLine_Containing <search string>`

Each command starts searching at the current position of the line pointer. If they find a matching line the line pointer is moved to it. Otherwise the line pointer keeps its position.

`<search string>` - as all other String references in the following commands - are String surrounded by single or double citation marks.

- `GoToTop`  
move the line pointer to the top line.

(when we count lines it has to be noted that this commands move the line pointer above the top line). We step any - positive or negative - number of lines through the file by

- `AdvanceLine [line count]`  
move the line pointer at `[line count]` lines forward or backward.
- `GoToBottom`  
Advancing to the bottom line

By the following command :

- `DeleteTheLine`  
we delete the line at which the line pointer is directed if there is such a line (if the line pointer has position top, nothing is deleted)
- `DeleteAllLines_StartingWith <search string>`  
deleting all lines which begin with `<search string>`
- `AddLine <line>` or `Add_Line <line>`  
The line is appended to the file.
- `InsertLine <line>` or `Insert_Line <line>`  
`<line>` is inserted at the position of the line pointer.
- `AppendLine <line>` or `Append_Line <line>`  
`<line>` is appended after the line at which the pointer is directed.
- `Append_File <file name>`  
reads the file and appends its lines to the edited file.
- `Subtract_File <file name>`  
removes the beginning lines of the edited file as long as they are identical with the lines of file `<file name>`.
- `SaveToFile <file name>`  
writes the edited lines as a file `<file name>`.
- `Sorted`  
causes that the edited lines are (ASCII) ordered.

### 8.5.3 Examples

For more examples, please check the opsi standard product *opsi-winst-test* and in this product the part `$Flag_winst_patch_text_file$ = "on"`

## 8.6 LinkFolder Sections

In a LinkFolder section start menus entries as well as desktop links are managed.

E.g. the following section creates a folder named "acrobat" in the common start menu (shared by all users):

```
[LinkFolder_Acrobat]
set_basefolder common_programs

set_subfolder "acrobat"
set_link
  name: Acrobat Reader
  target: C:\Programme\adobe\Acrobat\reader\acrord32.exe
  parameters:
  working_dir: C:\Programme\adobe\Acrobat\reader
  icon_file:
  icon_index:
end_link
```

As can be seen in the example, in a LinkFolder section the first thing to set is the virtual system folder on which the following statements shall operate:

**set\_basefolder** <system folder>

The predefined virtual system folders which can be used are:

*desktop, sendto, startmenu, startup, programs, desktopdirectory, common\_startmenu, common\_programs, common\_startup, common\_desktopdirectory*

The folders are *virtual* since the operating system (resp. registry entries) determine the real places of them in the file system. Second, we have to open a subfolder of the selected virtual folder:

**set\_subfolder** <folder path>

The subfolder name is to be interpreted as a path name with the selected virtual system folder as root. If some link shall be directly placed into the system folder we have to write

**set\_subfolder** ""

In the third step, we can start setting links. The command is a multi line expression starting with

**set\_link**

and finished by **end\_link**.

Between these lines the link parameters are defined in the following format:

```
set_link
name: [link name]
target: <complete program path>
parameters: [command line parameters of the program]
working_dir: [working directory]
icon_file: [icon file path]
icon_index: [position of the icon in the icon file]
end_link
```

The *target* name is the only essential entry. The other entries have default values:

- **name** defaults to the program name.
- **`parameters`** defaults to a empty string.
- **icon\_file** defaults to the *target*.
- **`icon\_index`** defaults to 0.

**Caution**

If the referenced target does not lie on an mounted share at the moment of link creation windows shortens its name to the 8.3 format.

Workaround:

Create a correct link when the share is connected.

Copy the ready link file to a location which exists at script runtime.

Let this file be the target.

- `delete_element <Linkname>`  
remove a link from the open folder.
- `delete_subfolder <Folderpath>`  
folder is removed from the base virtual folder

### 8.6.1 Examples

```
set $list2$ = createStringList ('common_startmenu', 'common_programs', 'common_startup', '
    common_desktopdirectory')
for $var$ in $list2$ do LinkFolder_Dummy

[LinkFolder_Dummy]
set_basefolder $var$
set_subfolder "Dummy"
set_link
    name: Dummy
    target: C:\Programme\PuTTY\putty.exe
    parameters:
    working_dir: C:\Programme\PuTTY
    icon_file:
    icon_index:
end_link
```

produces the following log:

```
Set $list2$ = createStringList ('common_startmenu', 'common_programs', 'common_startup', '
common_desktopdirectory')
retrieving strings from createStringList [switch to loglevel 7 for debugging]
(string 0)common_startmenu
(string 1)common_programs
(string 2)common_startup
(string 3)common_desktopdirectory

retrieving strings from $list2$ [switch to loglevel 7 for debugging]
(string 0)common_startmenu
(string 1)common_programs
(string 2)common_startup
(string 3)common_desktopdirectory

~~~~~ Looping through: 'common_startmenu', 'common_programs', 'common_startup', '
common_desktopdirectory'

Execution of LinkFolder_Dummy
Base folder is the COMMON STARTMENU folder
Created "Dummy" in the COMMON STARTMENU folder
```

```

ShellLink "Dummy" created

Execution of LinkFolder_Dummy
Base folder is the COMMON PROGRAMS folder
Created "Dummy" in the COMMON PROGRAMS folder
ShellLink "Dummy" created

Execution of LinkFolder_Dummy
Base folder is the COMMON STARTUP folder
Created "Dummy" in the COMMON STARTUP folder
ShellLink "Dummy" created

Execution of LinkFolder_Dummy
Base folder is the COMMON DESKTOPDIRECTORY folder
Created "Dummy" in the COMMON DESKTOPDIRECTORY folder
ShellLink "Dummy" created

~~~~~ End Loop

```

For more examples, please check the opsi standard product *opsi-winst-test* and in this product the part `$Flag_winst_link_folder$ = "on"`.

## 8.7 XMLPatch Sections

Today, the most popular way to keep configuration data or data at all is a file in XML document format. Its syntax follows the conventions as defined in the XML (or "Extended Markup Language") specification (<http://www.w3.org/TR/xml/>).

*opsi-winst* offers XMLPatch sections for editing XML documents.

With the actions defined for this section type *opsi-winst* can

- *select* (and optionally create) sets of elements of a XML document according to a path description
- *patch* all elements of a selected element set
- *return* the names and/or attributes of the selected elements to the calling section

### 8.7.1 Parameter

When calling an XMLPatch section the document path name is given as parameter, e.g.

```
XMLPatch_mozilla_mimetypes $mozillaprofilepath$ + "\mimetypes.rdf"
```

### 8.7.2 Structure of a XML Document

A XML document logically describes a "tree" which starting from a "root" - therefore named document root- grows into branches. Every branch is labelled a node. The sub nodes of some node are called children or child nodes of their parent node.

In XML, the tree is constructed from elements. The beginning of any element description is marked by a tag (similarly as in HTML) i.e. a specific piece of text which is set into a pair of angle brackets ("`<`" "`>`"), The end of the element description is defined by the the same tag text but now bracket by "`</`" and "`>`". If an element has no subordinated elements then there is no space needed between start tag and end tag. In this case the two tags can be combined to one with end bracket "`>`".

This sketch shows a simple "V"-tree - just one branching at the root level, rotated so that the root is top:

```

|      root node (level 0)
/ \    node 1 and node 2 both on level 1
. .    implicitly given end nodes below level 1

```

This tree could be described in XML in the following way:

```

<?xml version="1.0" ?>
<root>
  <node_level_1_no_1>
  </node_level_1_no_1>
  <node_level_1_no_2>
  </node_level_1_no_2>
</root>

```

The first line has to declare the XML version used. The rest of lines describe the tree.

So long the structure seems to be simple. But yet we have only "main nodes" each defining an element of the tree and marked by a pair of tags. But each main node may have subnodes of several kinds.

Of course, an element may have subordered elements, e.g. we may have subnodes A to C of node 1:

```

<node_level_1_no_1>
  <node_level_2_A>
  </node_level_2_A>
  <node_level_2_B>
  </node_level_2_B>
  <node_level_2_C>
  </node_level_2_c>
</node_level_1_no_1>

```

If there are no subordinated elements an element can have subordinated text. Then it is said that the element has a subordinated text node. Example

```

<node_level_1_no_2>hello world
</node_level_1_no_2>

```

A line break placed in the text node is now interpreted as part of the text where otherwise it is only a means of displaying XML structure. To avoid a line break belonging to "hello world" we have to write

```

<node_level_1_no_2>hello world</node_level_1_no_2>

```

Every element (no matter if it has subordinated elements or subordinated text) is constituted as a main node with specific tags. It can be further specified by attributes, so called attribute nodes. For example, there may be attributes "colour" or "angle" that distinguish different nodes of level 1.

```

<node_level_1_no_1 colour="green" angle="65"
</node_level_1_no_1>

```

For selecting a set of elements any kind of information can be used:

1. the element level,
2. the element names that are traversed when descending the tree (the "XML path"),
3. names and values of the used attributes,
4. the ordering of attributes,
5. the ordering of elements,

6. other relationships of elements,
7. the textual content of elements (resp. their subordinated text nodes).

In *opsi-winst*, selection based on criteria (1) to (3) and (7) is implemented

### 8.7.3 Options for Selection a Set of Elements

Before any operation on the contents of a XML file the precise set of elements has to be determined on which it will be operated. The set is constructed step by step by defining the allowed paths through the XML tree. The finally remaining end points of the paths define the selected set.

The basic *opsi-winst* command is

- OpenNodeSet

There two formats for defining the allowed paths a short and a long format .

**Explicit Syntax** The more explicit syntax may be seen in the following example (for a more complex example Section 10.4):

```
openNodeSet
  documentroot
  all_childelements_with:
    elementname:"define"
  all_childelements_with:
    elementname:"handler"
    attribute: extension value="doc"
  all_childelements_with:
    elementname:"application"
end
```

**Short Syntax** The same node set is given by the line:

```
openNodeSet 'define /handler value="doc"/application /'
```

In this syntax, the slash separates the steps into to the tree structure which are denoted in the more explicit syntax each by an own description.

**Selecting by Textual Content (only for explicit syntax)** Given the explicit syntax we may select elements by the textual content of elements:

```
openNodeSet
  documentroot
  all_childelements_with:
  all_childelements_with:
    elementname:"description"
    attribute:"type" value="browser"
    attribute:"name" value="mozilla"
  all_childelements_with:
    elementname:"linkurl"
    text:"http://www.mozilla.org"
end
```

**Parametrizing Search Strategy** In the exemplary descriptions of XML tree traversals there remain several questions.

- Shall an element be accepted if the element name and the listed attributes match but other attributes exist?
- Is the search meant to give one single result value, that is should the resulting element set have no more than one element (and otherwise, the XML file is to considered as erroneous)?
- Conversely, is it meant that a traversal shall at any rate lead to some result, i.e. do we have to create the element if no matching element exists?

To answer these questions explicitly there are parameters for the `OpenNodeSet` command. The following lines show the default settings which can be varied by changing the Boolean values:

```
- error_when_no_node_existing false
- warning_when_no_node_existing true
- error_when_nodccount_greater_1 false
- warning_when_nodccount_greater_1 false
- create_when_node_not_existing false
- attributes_strict false
```

With short syntax, parametrizing precedes the `OpenNodeSet` command and holds for all levels of the XML tree. With the explicit syntax the parameters may be set directly after the `OpenNodeSet` command or be newly set for each level. In particular the option „create when node not existing“ may be set for some levels but not for all.

#### 8.7.4 Patch Actions

There exists a bundle of commands which operate on a selected element set

- for setting and removing attributes
- for removing elements
- for text setting..

In detail:

- **SetAttribute** "attribute name" value="attribute value"  
sets the specified attribute for each element in the opened set to the specified value. In the attribute does not exist it will be created.  
Example: `SetAttribute "name" value="OpenOffice Writer"`

On the contrary, the command

- **AddAttribute** "attribute name" value="attribute value"  
sets the specified attribute only to the specified value if it does not exists beforehand. An existing attribute keeps its value. E.g. the command  
`AddAttribute "name" value="OpenOffice Writer"`  
would not overwrite the value if there was named another program before.

By

\* **DeleteAttribute** "attribute name"  
we remove the specified attribute from each element of the selected element set.

The command

\* **DeleteElement** "element name"  
removes all elements with main node name (tag name) element name from the opened element set.

Finally there exist two commands for setting resp. adding text nodes.:

- `SetText "Text"`

and

- `AddText "Text"`

A. g.

```

SetText "rtf"
transforms the element
<fileExtensions>doc<fileExtensions>
into
<fileExtensions>rtf<fileExtensions>

```

By

```
SetText ""
```

we remove the text node completely.

The variant

```
AddText "rtf"
```

sets the text only if there was no text node given.

### 8.7.5 Returning Lists to the Caller

A XMLPatch section may return the retrieved informations to the calling primary section. The result always is a String list, and to get it, the call must done via the String list function `getReturnListFromSection`. E.g. we may have the following String list setting in an Actions section where we use a XMLPatch\_mime section

```

DefStringList $list1$
set $list1$=getReturnListFromSection ('XMLPatch_mime "c:\mimetypes.rdf"')

```

Inside the XMLPatch section we have `return` commands that determine the content of returned String list:

- `return elements+` fills the selected elements completely (element name and attributes) into the return list.
- `return attributes` produces a list of the attributes.
- `return elementnames` produces a list of the element names.
- `return attributenames` gives a list only of the attribute names.
- `return text` list all textual content of the selected elements.
- `return counting` gives a report with numerical informations: line 0 contains the number of selected elements, line 1 the number of attributes.

### 8.7.6 Examples

For further examples see the product *opsi-winst-test* especially the sector with `$Flag_winst_xml$ = "on"`

## 8.8 ProgmanGroups Sections

This section type is deprecated.

## 8.9 WinBatch-Sections

In a WinBatch section any windows executable can be started.

E.g, we may start some existing setup program by the following line in a WinBatch section

```
[winbatch_install]
"%scriptpath%\setup.exe"
```

It is deprecated but still supported that you may call – as from Windows explorer – a file of any type for which a program is registered.

### 8.9.1 Call Parameter (Modifier)

There are several parameters of the WinBatch call which determine if (or how long) *opsi-winst* shall wait for the started programs returning.

- **/WaitOnClose**  
Is the default  
*opsi-winst* waits for every initiated process to come back.
- **/LetThemGo**  
This is the contrary to **/WaitOnClose**. It is used if *opsi-winst* shall proceed while the started processes run in their own threads.
- **/WaitSeconds [number of seconds]**

If we do the call with parameter **/WaitSeconds [number of seconds]** then *opsi-winst* is waiting the specified time before proceeding. In the default configuration we additionally wait for the started programs returning. If we combine the parameter with the option **/LetThemGo** then *opsi-winst* continues processing when the waiting time is finished.

- **/WaitForWindowAppearing [window title]**  
resp.  
**/WaitForWindowVanish [window title]**  
Both are deprecated. Please use **/WaitForProcessEnding**

The first option means that *opsi-winst* waits until any process lets pop up a window with title window title. With the second option *opsi-winst* is waiting as long as a certain window (1) appeared on the desktop and (2) disappeared again.

CAUTION: These options only know windows of 32-bit programmes

- **/WaitForProcessEnding <program name>**  
Waits for the ending of the process with the name <program name>.  
Should be combined with **/TimeOutSeconds**.
- **/TimeOutSeconds <seconds>**  
A timeout setting. After <seconds> the process will be canceled.  
May not be used without a waiting condition (e.g. **/WaitForProcessEnding**)  
Example:

```
Winbatch_uninstall /WaitForProcessEnding "uninstall.exe" /TimeOutSeconds 20
[Winbatch_uninstall]
"%ScriptPath%\uninstall_starter.exe"
```

- `getLastExitCode`

The String function `getLastExitCode` gives access to the `ExitCode` – or `ErrorLevel` – of the last process call in the preceding WinBatch section.

## 8.9.2 Examples

For further examples see the product *opsi-winst-test* especially the sector with `$Flag_winst_winbatch$ = "on"`

## 8.10 DOSBatch/DosInAnIcon (ShellBatch/ShellInAnIcon) Sections

Via DOSBatch (also called ShellBatch) sections a *opsi-winst* script uses Windows shell scripts for tasks which cannot be fulfilled by internal commands or for which already a batch script solution exists.

*opsi-winst* waits until the DOS-batch ends, before it is proceeding with the next script-section.

A DOSBatch section is simply processed by writing the lines of the sections into the file `__winst.bat` in `c:\tmp` and then calling this file in the context of a `cmd.exe` shell. This explains that a `DosBatch` section may contain all Windows shell commands can be used.

Compared with calling a `cmd`-file in a WinBatch section, the DOSBatch section presents certain advantages:

- *opsi-winst* variables or constants in the section can be easily used because they will be substituted before execution.
- The output of the scripts is written to the log file.
- The output of the shell commands can be captured by using the String list function.

The section type *DOSInAnIcon* or *ShellInAnIcon* is identical to *DOSBatch* regarding syntax and execution method but has a different appearance:

For *DOSInAnIcon*, a shell process is created with view set to minimized. That has the consequence that it is executed "in an icon". No command window appears, user interaction is suppressed.



### Caution

Don't use commands that wait for user interaction.

---

### 8.10.1 Parameter

There are two kinds of parameters which may be passed to the section call:

- Parameters which are directly passed to the called batch file.
- Parameter which modify the way *opsi-winst* will handle the section.

The calling syntax is:

```
Sektionsname [batch parameter] [winst [modifier]]
```

Possible `winst` modifier are (since 4.11.1):

- `/32bit`
- `/64bit`

- /Sysnative

These winst modifier have to be separated by the key word **winst** from the other parameters.

Other parameters of a DosBatch section are directly passed as quasi command line parameters to the Windows shell script.

E. g. we may call DosBatch\_1 in Actions section to get a "Hello World" from the DOS echo command:

```
[Actions]
DosBatch_1 today we say "Hello World"

[DosBatch_1]
@echo off
echo %1 %2 %3 %4
pause
```

the execution of the Dos-Batch command echo with parameters *today we say "Hello World"*.

The next example will be on a 64 bit system executed in a 64 bit cmd.exe and gives the output *today we say*:

```
[Actions]
DosBatch_1 today we say winst /64bit

[DosBatch_1]
@echo off
echo %1 %2 %3 %4
pause
```

## 8.10.2 Catch the output

The output of the shell commands can be captured by using the string list function `getOutputStreamFromSection()` from the *opsi-winst*-scripts main-section see also:

Section 7.4.4).

If the return list shall be evaluated programmatically it is advised to use the @ prefix of commands. Such we suppress the repetition of the command line in the output which may different formats dependent on system configurations.

## 8.10.3 Examples

For further examples see the product *opsi-winst-test* and there look at `$Flag_winst_dos$ = "on"`

## 8.11 Registry-Sections

By a Registry section call we can create, patch and delete entries in the Windows registry. As usual, *opsi-winst* logs every operation in detail as long as logging is not turned off.

### 8.11.1 Examples

Let us set some registry variables by a call to the section Registry\_TestPatch where the section is given by

```
[Registry_TestPatch]
openkey [HKEY_Current_User\Environment\Test]
set "Testvar1" = "c:\rutils;%Systemroot%\hey"
set "Testvar2" = REG_DWORD:0001
```

For further examples see the product *opsi-winst-test* and there look at `$Flag_subregistry$ = "on"`

### 8.11.2 Call Parameters

- The standard call of a Registry section has no parameters. This is sufficient as long as the operations aim at the standard registry of a Windows system and all entries can be defined using a globally defined registry path.
- `/AllNTUserDats`  
*opsi-winst* also offers that the patch commands of a Registry section are automatically executed "for all users" which are locally defined. I.e. the patches are made for all user branches of the local registry. This interpretation of the section is evoked by the parameter `/AllNTUserDats`

Further parameters control which syntactical variant of the Registry section shall be valid:

- `/regedit`  
The parameter `/regedit` declares that the syntax corresponds the export file syntax of the Windows Registry Editor regedit. Such, the lines of a regedit export file may directly be used as a Registry resp. the file itself can serve as an external section (see Section 8.11.5).
- `/addReg`  
Similarly, the parameter `/addReg` declares that the Registry section syntax is that of an inf-file (as used e.g. for driver installations) (see Section 8.11.6).

These not *opsi-winst* specific syntactical variants are not defined in this manual since they usually will be generated programmatically.

There are also the Options:

- `/32Bit`
- `/64Bit`
- `/SysNative`

which manipulate the *registry write redirection* on 64 Bit systems. (see Chapter 9).

### 8.11.3 Commands

The default syntax of a Registry section is oriented at the command syntax of other patch operations in *opsi-winst*.

There exist the following commands:

- `OpenKey`
- `Set`
- `Add`
- `Supp`
- `GetMultiSZFromFile`
- `SaveValueToFile`
- `DeleteVar`
- `DeleteKey`
- `ReconstructFrom`

- **Flushkey**

In detail:

- **OpenKey** <registry key>  
opens the specified key for reading and (if the user has the necessary privileges) for writing. If the key does not exist it will be created.

The registry key is denoted by a registry path name. Under regular circumstances it starts with one of the "high keys" which build the top level of the registry tree data structure (above the "root"). These are: *HKEY\_CLASSES\_ROOT*, *HKEY\_CURRENT\_USER*, *HKEY\_LOCAL\_MACHINE*, *HKEY\_USERS*, *HKEY\_CURRENT\_CONFIG* which may optionally be written as *HKCR*, *HKCU*, *HKLM*, *HKU*.

In *opsi-winst* syntax of the registry path name, the elements of a path are separated by single backslashes.

All other commands operate on an opened registry key.

- **Set** <varname> = <value>  
sets the registry variable <varname> to value <value>. <varname> as well as <value> are strings and have to be enclosed in citations marks. A non-existing variable will be created. The empty variable "" denotes the standard entry of a registry key.

If some registry variable shall be created or set which has not the default type Registry-String (REG\_SZ) we have to use the extended variant of the **Set** command:

- **Set** <varname> = <registry type>:<value>  
sets the registry variable <varname> to value <value> of type <registry type>. The following registry types are supported:

**REG\_SZ**

(string)

**REG\_EXPAND\_SZ**

(a string containing substrings which the operating system shall expand e.g.)

**REG\_DWORD**

(integer values)

**REG\_BINARY**

(binary values usually given as two-digit hex numbers 00 01 02 .. 0F 10 .., notiert)

**REG\_MULTI\_SZ**

(string value arrays, in *opsi-winst* we have to use "|" as separator)

An example for setting a REG\_MULTI\_SZ:

```
set "myVariable" = REG_MULTI_SZ:"A|BC|de"
```

To construct a multistring we may put the strings as lines in a file and read it using `GetMultiSZFromFile` (cf. below).

- **Add** <varname> = <value>  
resp.  
**Add** <varname> = <registry type> <value>  
are analogous to the **Set** commands with the difference that entries are only added but values of existing variables not changed.
- **Supp** <varname> <list separator> <supplement>  
This command interprets the string value of variable <varname>, a list of values separated by <list separator> and adds the string <supplement> to this list (if it not already contained). If <supplement> contains the <list separator> it is split into single strings, and the procedure is applied to each single string.  
A typical use is adding entries to a path variable (which is defined in the registry).  
**Supp** keeps the original string variant (REG\_EXPAND\_SZ or REG\_SZ).

Example:

The environment Path is determined by the value for the variable Path as defined inside the registry key

+ `KEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Environment`

+ To add some entries to the path definition we have to get access to this key via an OpenKey. Then we can apply e.g.

+ `supp "Path" ; "C:\utils;%JAVABIN%"`

+ in order to supplement the path by `"C:\utils"` and `"%JAVABIN%"`.

+ (Windows expands %JAVABIN% to the correct path name if %JAVABIN% exists as variable and the String is a REG\_EXPAND\_SZ.)

+ Whom read the old value of Path from the environment variable, write this value to the registry value - and are then able to work with the registry variable:

+

```
[Actions]
DefVar $Path$
set $Path$ = EnvVar ("Path")
Registry_PathPatch

[Registry_PathPatch]
openkey [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\control\Session Manager\Environment]
set "Path"="$Path$"
supp "Path"; "c:\orawin\bin"
```

+ CAUTION: The environment variable gets a changed value after a reboot.

- `GetMultiSZFromFile <varname> <filename>`  
reads the lines of a file and puts them together building a Multistring.
- `SaveValueToFile <varname> <filename>`  
exports the referred (String or MultiSZ) value as file <filename> lines (each String forming a line).
- `DeleteVar <Varname>`  
removes the entry with variable <varname> from the opened key.
- `DeleteKey <Registrieschlüssel>`  
deletes the registry key recursively including all subkeys and contained variables. The registry key is defined as for OpenKey.

Example:

```
[Registry_Keydel]
deletekey [HKCU\Environment\subkey1]
```

- `ReconstructFrom <filename>`  
(deprecated)
- `FlushKey`  
ensures that all entries of a key are saved on hard drive, not only in memory (is automatically done when closing a key, therefore in particular when a registry section is left).

#### 8.11.4 Registry Sections to Patch *All NTUser.dat*

A Registry section called with parameter `/AllNTUserdat`s is executed for every local user.

To this end, for all local users (as permanent storage for the registry branch *HKEY\_Users*) the files *NTUser.dat* are searched one by one and temporarily loaded into a subkey of some registry branch. The commands of the registry section are executed for this subkey, then the subkey is unloaded. As result, the stored *NTUser.dat* is changed.

The mechanism does not work for a logged in user. For, his *NTUser.dat* is already in use, and the request to load it produces an error. To do the changes for him as well, the commands of the Registry additionally are executed on the *HKEY\_Users* branch for the logged in user.

There is a *NTUser.dat* for *Default User* which serves as template for newly created users in the future. Therefore the patches are prepared for them as well.

The Registry section syntax remains unchanged. But the key paths are interpreted relatively. This means **leave away the main key**

In the following example the registry entry for variable *FileTransferEnabled* is de facto set for all *HKEY\_Users\XX\Software...* successive for all XX (all users) on the machine:

```
[Registry_AllUsers]
openkey [Software\ORL\WinVNC3]
set "FileTransferEnabled"=reg_dword:0x00000000
```

Since *opsi-winst* version 4.11.2 you may give the root key *HKEY\_CURRENT\_USER* at the *openkey* command. Example:

```
[Registry_AllUsers]
openkey [HKEY_CURRENT_USER\Software\ORL\WinVNC3]
set "FileTransferEnabled"=reg_dword:0x00000000
```

This has the following advantages:

- The syntax is easier to understand.
- The same registry section may be used with */AllNtuserdats* and in a *userLoginScript*

### 8.11.5 Registry Sections in Regedit Format

If a Registry section is called with parameter */regedit* the section is not expected in *opsi-winst* standard format but in the format as produced by the Windows regedit tool. The export files generated by regedit have - not regarding the head line - ini file format. Example:

```
REGEDIT4

[HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org]

[HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org\general]
"bootmode"="BKSTD"
"windomain"=""
"opsiconf"=dword:00000001

[HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org\shareinfo]
"user"="pcpatch"
"pcpatchpass"=""
"depoturl"="\\\\bonifax\opt_pcb\install"
"configurl"="\\\\bonifax\opt_pcb\pcpatch"
"utilsurl"="\\\\bonifax\opt_pcb\utils"
"utilsdrive"="p:"
"configdrive"="p:"
"depotdrive"="p:"
```

The sections denote registry keys to be opened. Each line describes some variable setting like the set command in a *opsi-winst* registry section.

But, we cannot really have an internal *opsi-winst* section that is constructed from another sections. Therefore Registry section with parameter */regedit* can only be given as external section or by the function call *loadTextFile*, e.g.

```
registry "%scriptpath%/opsiorgkey.reg" /regedit
```

With Windows XP the registry editor regedit does not produce Regedit4-Format but a new format that is indicated by the head line

*"Windows Registry Editor Version 5.00"*

In this format, Windows offers some additional value types. But more important, the export file is now generated in Unicode. *opsi-winst* sections processing is based on Delphi libraries which use 8 bit Strings. To work with a regedit 5 export the coding therefore has to be converted. This can be done manually, e.g. by a suitable editor. But we may also feed the original file to *opsi-winst* using the String list function `loadUnicodeTextFile`. E.g., if `printerconnections.reg` be a unicode based export, we can call regedit in the following form which does the necessary code conversion on the fly:

```
registry loadUnicodeTextFile("%scriptpath%/opsiorgkey.reg") /regedit
```

A registry patch using regedit format can as well be executed "for all NT users" similarly as the common *opsi-winst* registry section. That is, a path like `[HKEY_CURRENT_USER\Software\ORL]` is to be replaced by the relative `[Software\ORL]`.

### 8.11.6 Registry Sections in AddReg Format

A Registry section can be called with parameter `/addReg`. Then its syntax follows the principles of the `[AddReg]` sections in `inf` files as used e.g. for driver installations.

E.g.:

```
[Registry_ForAcroread]
HKCR, ".fdf", "", 0, "AcroExch.FDFDoc"
HKCR, ".pdf", "", 0, "AcroExch.Document"
HKCR, "PDF.PdfCtrl.1", "", 0, "Acr"
```

## 8.12 OpsiServiceCall Sections

This type of section allows to retrieve information – or set data – via the opsi service. There are three options for determining a connection to an opsi service:

- Per default it is assumed that the script is executed in the standard opsi installation environment. I.e., we already have a connection to an opsi service and can use it.
- We set the url of the service to which we want to connect as a section parameter and supply as well the required username and password as section parameters.
- We demand an interactive login to the service (predefining only the service url and, optionally, the user name).

Retrieved data may be returned as a String list and then used for scripting purposes.

### 8.12.1 Call Parameters

The call parameters determine which opsi service will be addressed and set the connection parameters if needed.

Connection parameters can be defined via

- `/serviceurl <url to the opsi web service>`

- `/username` <web service user name>
- `/password` <web service user password>

If these parameters, at least the `serviceurl`, are given *opsi-winst* tries to open a connection to an opsi service which has the url.

The additional option

- `/interactive`

raises an interactive connect. The user will be asked for confirming the connection data and supplying the password. Of course, this option cannot be used in scripts which shall be executed fully automatically.

If no connection parameters are supplied *opsi-winst* assumes that an existing connection shall be reused.

If no connection parameters are given and the `interactive` option is not specified (neither at this call nor at a call earlier in the script) it is assumed that we are in a standard opsi boot process and, already having a connection to an opsi service, we try to address it.

- `/preloginservice`  
In the case that we had a connection to a secondary opsi service we may (re)set the connection to the standard opsi service via the option
- `/opsiclientd` //since 4.11.2.1  
calls the localhosts `opsiclientd`

## 8.12.2 Section Format

An `opsiServiceCall`, which uses an existing connection to an opsi-service, is defined by its method name and a list of parameters.

Both are defined in the section body. It has format

```
"method":<method name>
"params": [
  <params>
]
```

*params* is a (possibly empty) list of strings (comma-separated). Use the parameters as required by the specified method.

E.g. we may have a section call where the required `methodname` and the (empty) list of parameters is set:

```
[opsiservicecall_clientIdsList]
"method": "getClientIds_list"
"params": []
```

The section call produces the list of names (IDs) of all local opsi clients. If the list shall be exploited for other than test purposes the section call can be used in a string list expression: E.g.:

```
DefStringList $result$
Set $result$=getReturnListFromSection("opsiservicecall_clientIdsList")
```

The usage of `GetReturnListFromSection` is documented in the string list function chapter of this manual (see Section 7.4.5).

A hash – in this case a string list, where each item is a pair `name=value` – is produced by the following opsi service call:

```
[opsiservicecall_hostHash]
"method": "getHost_hash"
"params": [
    "pcbon8.uib.local"
]
```



### Caution

The sections `opsiservicecall` are developed for opsi 3.x methods. For opsi 4.x methods they are often not suitable. For example `*_getIdsents` calls are possible, `*_getObjects` calls are not possible.

## 8.12.3 Examples

For further examples watch the product `opsi-winst-test` and there especially `$Flag_winst_opsiServiceCall$ = "on"`

## 8.13 ExecPython Sections

*ExecPython* sections are basically Shell-Sections (like *DosInAnIcon*) which call the – on the system installed – python script interpreter. It takes the section content as python script, and the section call parameter as parameters for the script.

Example:

The following example demonstrates a `execPython` call with a list of parameters for that are printed by the python commands.

The call may look like

```
execpython_hello -a "option a" -b "option b" "there we are"

[execpython_hello]
import sys
print "we are working in path: ", a
if len(sys.argv) > 1 :
    for arg in sys.argv[1:] :
        print arg
else:
    print "no arguments"

print "hello"
```

The print command output will be caught and written to the log file. So we get in the log

```
output:
-----
-a
option a
-b
option b
there we are
    hello
```

Observe that the loglevel must be set at least to info (that is 1) if these outputs shall really find their way to the log file.

### 8.13.1 Interweaving a Python Script with the opsi-winst Script

An execPython section is integrated with the surrounding *opsi-winst* script by four kinds of shared data:

- A parameter list is transferred to the python script.
- Everything that is printed by the python script is written into the *opsi-winst* log.
- The *opsi-winst* script substitution mechanism for constants and variables when entering a section does its expected work for the execPython section.
- The output of an execPython section can be caught into a stringlist and then used in the ongoing *opsi-winst* script.

An example for the first two ways of interweaving the python script with the *opsi-winst* script is already given above. We extend it to retrieve the values of some *opsi-winst* constants or variables.

```
[execpython_hello]
import sys
a = "%scriptpath%"
print "we are working in path: ", a
print "my host ID is ", "%hostID%"
if len(sys.argv) > 1 :
    for arg in sys.argv[1:] :
        print arg
else:
    print "no arguments"

print "the current loglevel is ", "$loglevel$"
print "hello"
```

Of course, the *\$loglevel\$* variable has to be set beforehand in the Actions section:

```
DefVar $LogLevel$
set $loglevel$ = getLogLevel
```

Finally, in order to being able to use of some results of the section output, we produce it into a stringlist variable by calling the execPython section in the following way:

```
DefStringList pythonresult
Set pythonResult = GetOutputStreamFromSection('execpython_hello -a "opt a"')
```

### 8.13.2 Examples

For further examples watch the product *opsi-winst-test* and there especially *\$Flag\_compare\_to\_python\$ = "on"*

## 8.14 ExecWith Sections

*ExecWith* sections are more general than *ExecPython* or *DosBatch* sections: Which program interprets the section lines given is determined by a parameter of the section call.

E.g, if we have some call

```
execPython_hello -a "hello" -b "world"
```

where

```
-a "hello" -b "world"
```

are parameters that are passed to the python script we get the following completely equivalent ExecWith call:

```
execWith_hello "python" PASS -a "hello" -b "world" WINST /EscapeStrings
```

The option `/EscapeStrings` is automatically used in an ExecPython section and means that backslashes in String variables and constants are duplicated before interpretation by the the called program.

### 8.14.1 Call Syntax

In general, we have the call syntax:

```
ExecWith_SECTION PROGRAM PROGRAMPARAS pass PASSPARAS winst WINSTOPTS
```

Each of the expressions *PROGRAM*, *PROGRAMPARAS*, *PASSPARAS*, *WINSTOPTS* may be an arbitrary String expression, or just a String constant (without citation marks).

The key words PASS and WINST may be missing if the respective parts do not exist.

There are two *opsi-winst* options recognized:

- `/EscapeStrings`
- `/LetThemGo`

Like with ExecPython sections, the output of an ExecWith section may be captured into a String list via the function `getOutputStreamFromSection`.

The first one declares that the backslash in *opsi-winst* variables and constants is C-like escaped. The second one has the effect (as for *winBatch* calls) that the called program starts its work in new thread while *opsi-winst* is continuing to interpret its script.

### 8.14.2 More Examples

The following call is meant to refer to a section which is an autoit3 script that waits for some upcoming window (therefore the option `/letThemGo` is used) in order to close it:

```
ExecWith_close "%SCRIPTPATH%\autoit3.exe" WINST /letThemGo
```

A simple

```
ExecWith_edit_me "notepad.exe" WINST /letThemGo
```

calls notepad and opens the section lines in it (but without any line that is starting with a semicolon since *opsi-winst* regards such lines as comments and eliminates them before handle).

For further examples watch the product *opsi-winst-test* and there especially `$Flag_autoit3_test$ = "on"`

## 8.15 LDAPsearch Sections

A LDAPsearch section defines a search request to a LDAP directory, executes it and receives (and possibly caches) the response.

Before dwelling on the *opsi-winst* commands we do some explanations.

In subsection we give an example of the most probable usage of a LDAPsearch. The following subsections describe the syntax

### 8.15.1 LDAP – Protocol, Service, Directory

LDAP, the "Lightweight Directory Access Protocol", is, as the name indicates, a defined way of communication to a directory. This directory is (or may be) hierarchically organized. That is, the directory is a hierarchical data base, or a tree of content.

A **LDAP service** implements the protocol. A directory that can be accessed via a LDAP service is called a **LDAP directory**.

For instance, let's have a look at some part of the LDAP directory tree of an opsi server with LDAP backend (as shown by the Open Source tool JXplorer):

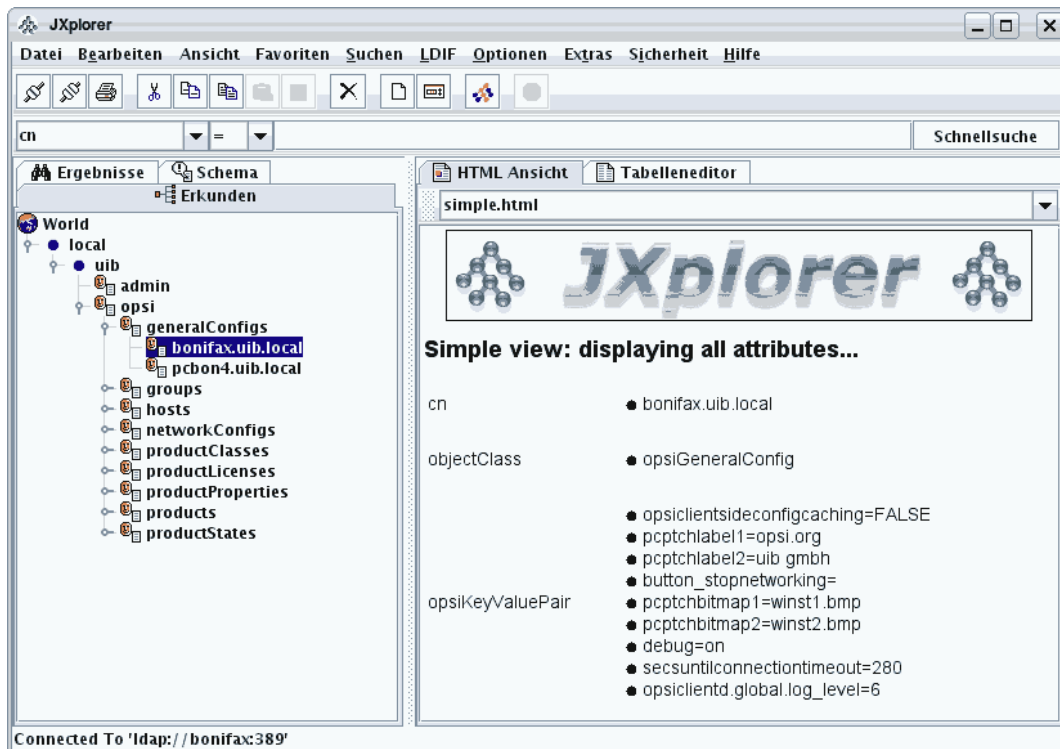


Figure 8.1: View of some part of an opsi LDAP tree

A **LDAP search request** is a query to a LDAP directory via a LDAP service. The response returns some content from the directory.

Basically the search request has to describe the path in the directory tree which leads to the interesting piece of information. The path is the **distinguished name** (dn), composed of the names of the nodes (the "relative distinguished names"), which build the path, for instance:

*local/uib/opsi/generalConfigs/bonifax.uib.local*

Since each node is conceived as an instance of some structural object class, the path description is usually given in the following form: with indication of the classes (and starting with the last path element) :

*cn=bonifax.uib.local,cn=generalConfigs,cn=opsi,dc=uib,dc=local*

The path in a query is not necessarily "complete", and not leading to a unique leaf of the tree. On the contrary, partial paths are common.

But even if the path descends to a unique leaf, the leaf may contain several values. Each node of the tree has one or more classes as attribute types. To each one or may values may be associated.

For a given query path, we therefore may be interested

1. in the node set whose elements – the so called LDAP objects – match the given path,
2. the set of attributes that belong the nodes,
3. and the values that are associated to all of them.

Obviously, handling the amount of possibly returned response information is the main challenge when dealing with LDAP searches.

The following section shows the documentation of a LDAPsearch roughly corresponding to the screenshot above as executed by *opsi-winst*.

Example

Using the *opsi-winst* section `ldapsearch_generalConfigs`:

```
[ldapsearch_generalConfigs]
targethost: bonifax
dn: cn=generalConfigs,cn=opsi,dc=uib,dc=local
```

we will get a answer like this:

```
Result: 0
Object: cn=generalConfigs,cn=opsi,dc=uib,dc=local
Attribute: cn
          generalConfigs
Attribute: objectClass
          organizationalRole
Result: 1
Object: cn=pcbon4.uib.local,cn=generalConfigs,cn=opsi,dc=uib,dc=local
Attribute: cn
          pcbon4.uib.local
Attribute: objectClass
          opsiGeneralConfig
Attribute: opsiKeyValuePair
          test2=test
          test=a b c d
Result: 2
Object: cn=bonifax.uib.local,cn=generalConfigs,cn=opsi,dc=uib,dc=local
Attribute: objectClass
          opsiGeneralConfig
Attribute: cn
          bonifax.uib.local
Attribute: opsiKeyValuePair
          opsiclientsideconfigcaching=FALSE
          pcptchlabel1=opsi.org
          pcptchlabel2=uib gmbh
          button_stopnetworking=
          pcptchbitmap1=winst1.bmp
          pcptchbitmap2=winst2.bmp
          debug=on
          secsuntilconnectiontimeout=280
          opsiclientd.global.log_level=
```

There are several *opsi-winst* options to manage and reduce the complexity of the evaluation of such responses.

## 8.15.2 LDAPsearch Call Parameters

Two kinds of LDAPsearch parameters,

- cache options
- output options

are defined for the call of LDAPsearch section.

The *cache options* are:

- `/cache`
- `/cached`
- `/free`
- (no cache option)

If there is no cache option specified, the response of the LDAP search request is not saved for further usages.

By the `/cache` option, the response is cached for further evaluations, the `/cached` option refers to the last cached response which is reused instead of starting a new search, the `/free` option clears the cache explicitly (may only be useful for searches with extreme large responses).

The *output options* are:

- `/objects`
- `/attributes`
- `/values`
- (no output option)

The output options determine the String list that is produced when a LDAPsearch section is called via `getReturnlistFromSection`:

- If no output option is specified the returned list is the complete LDAP response.
- The options `objects`, `attributes` and `values` restrict the output to object, attribute or value lines of the LDAP response respectively.

Observe that in the produced lists the object an attribute belongs to is only identifiable if only one object is returned in the object list, and likewise the object and the attribute to which a value is subsumed are only identifiable if there is only attribute remaining in the attributes list.

Such the proceeding is, that the LDAPsearch is specified up to that degree, that at most one object and one attribute is returned. This can be checked by a count call on the objects and the attributes return list. Then any value found belongs to the dn and the attribute specified.

The repeated utilization of the same LDAP response can be done without relevant time costs by using the `cache/cached` options.

### 8.15.3 How to Narrow the Search

An example may show how we can narrow the search to pin down a specific result from a LDAP directory.

We start with the call of `ldapsearch_generalConfigs` as above, only adding the cache parameter.

```
ldapsearch_generalconfigs /cache
```

executes the query and caches the response for further utilization.

Then, the call

```
getReturnlistFromSection("ldapsearch_generalconfigs /cached /objects")
```

produces the list

```
cn=generalconfigs,cn=opsi,dc=uib,dc=local
cn=pcbon4.uib.local,cn=generalconfigs,cn=opsi,dc=uib,dc=local
cn=bonifax.uib.local,cn=generalconfigs,cn=opsi,dc=uib,dc=local
```

If we narrow the tree selection by

```
[ldapsearch_generalConfigs]
targethost: bonifax
dn: cn=bonifax.uib.local,cn=generalConfigs,cn=opsi,dc=uib,dc=local
```

and start again, then in the objects list, we indeed retain just

```
cn=bonifax.uib.local,cn=generalconfigs,cn=opsi,dc=uib,dc=local
```

The corresponding attributes list contains three elements:

```
objectclass
cn
opsikeyvaluepair
```

In order to get the values associated to a single attribute we have to confine the query once more:

```
[ldapsearch_generalConfigs]
targethost: bonifax
dn: cn=bonifax.uib.local,cn=generalConfigs,cn=opsi,dc=uib,dc=local
attribute: opsiKeyValuePair
```

The result now produced is an attributes list containing only one element. The corresponding values list looks like

```
opsiclientsideconfigcaching=false
pcptchlabel1=opsi.org
pcptchlabel2=uib gmbh
button_stopnetworking=
pcptchbitmap1=winst1.bmp
pcptchbitmap2=winst2.bmp
debug=on
secsuntilconnectiontimeout=280
opsiclientd.global.log_level=6
```

There are no LDAP means to reduce this result furthermore!

(But the *opsi-winst* function `getValue (key, list)` (cf. Section 7.4.4) may help in this case: E.g. `getValue ("secsuntilconnectiontimeout", list)` would produce the requested number).

By the function `count (list)` we can check if we succeeded with the narrowing of the search request. In most circumstances, we would like that its result be "1".

#### 8.15.4 LDAPsearch Section Syntax

A LDAPsearch section comprises the specifications:

- **targethost:**  
The server hosting the LDAP directory (service) must be named.
- **targetport:**  
If the port of the LDAP service is not the default (389), it can be declared at this place. If the specification is missing, the default port is used.
- **dn:**  
Here, the distinguished name, the "search path", for the search request can be given.

- **typesonly:**  
Default "false", that is, values are retrieved.
- **filter:**  
A filter for LDAP search has a LDAP specific syntax that is not checked by *opsi-winst*. Default is "(objectclass=\*)"
- **attributes:**  
A comma separated list of attribute names may be given. The default is to take any attribute.

### 8.15.5 Examples

A short and rather realistic example shall end this section:

*\$founditems\$* be a StringList variable and *\$opsiClient\$* a String variable. The call of *getReturnlistFromSection* fetches the results of the section *ldapsearch\_hosts*. The following code fragment returns the unique result for *\$opsiDescription\$* if it exists. It reports an error if the search produces an unexpected result:

```
set $opsiClient$ = "test.uib.local"
set $founditems$ = getReturnlistFromSection("ldapsearch_hosts /values")

DefVar $opsiDescription$
set $opsiDescription$ = ""
if count(founditems) = "1"
  set $opsiDescription$ = takeString(0, founditems)
else
  if count(founditems) = "0"
    comment "No result found")
  else
    logError "No unique result for Ldapsearch for client " + $opsiClient$
  endif
endif

[ldapsearch_hosts]
targethost: opsiserver
targetport:
dn: cn=$opsiClient$,cn=hosts,cn=opsi,dc=uib,dc=local
typesOnly: false
filter: (objectclass=*)
attributes: opsiDescription
```

For further examples watch the product *opsi-winst-test* and there especially *\$Flag\_winst\_ldap\_search\$ = "on"*

## Chapter 9

# 64 Bit Support

The *opsi-winst* is a 32 bit program. In order to make it easy for 32 bit programs to run on 64 bit systems there are special 32 bit areas in the registry as well in the file system. Some accesses from 32 bit programs will be redirected to these special areas to avoid access to areas that reserved for 64 bit programs.

A access to `c:\windows\system32` will be redirected to `c:\windows\syswow64`

But a access to `c:\program files` will be **not** redirected to `c:\program files (x86)`

A registry access to `[HKLM\software\opsi.org]` will be redirected to `[HKLM\software\wow6432node\opsi.org]`

Therefore *opsi-winst* installs as 32 bit program scripts, that run on 32 bit system fine, on 64 bit system correct without any change.

For the installation of 64 bit programs some constants like `%ProgramFilesDir%` returns the wrong values. Therefore we have since *opsi-winst* 4.10.8 some new features:

Normally you may (and should) tell explicit to which place you want to write or from where you want to read. Here we have three variants:

### 32

explicit 32 bit

### 64

explicit 64 bit; if not on a 64 bi system like *sysnative*

### SysNative

according to the architecture on which the script runs

Following this idea, we have some additional constants:

Table 9.1: Constants

Constant	32 Bit	64 Bit
<code>%ProgramFilesDir%</code>	<code>c:\program files</code>	<code>c:\program files (x86)</code>
<code>%ProgramFiles32Dir%</code>	<code>c:\program files</code>	<code>c:\program files (x86)</code>
<code>%ProgramFiles64Dir%</code>	<code>c:\program files</code>	<code>c:\program files</code>
<code>%ProgramFilesSysnativeDir%</code>	<code>c:\program files</code>	<code>c:\program files</code>

### `%ProgramFilesDir%`

you should avoid this in future. . .

**%ProgramFiles32Dir%**

should be used in the context of installing 32 bit Software.

**%ProgramFiles64Dir%**

should be used in the context of installing 64 bit Software.

**%ProgramFilesSysnativeDir%**

should be used if you need architecture specific information

For a reading access to the different areas of registry and filesystem we have now the following new functions:

- GetRegistrystringValue32
- GetRegistrystringValue64
- GetRegistrystringValueSysNative
- FileExists32
- FileExists64
- FileExistsSysNative

A simple call to Registry-section results in writing to the 32 bit registry regions. Also a simple call to Files-section results in writing to the 32 bit file system regions.

For *Registry* and *Files* section we have now the additional calling options:

- /32Bit  
This is the default. Any access will be redirected to the 32 bit regions.
- /64Bit  
Any access will be redirected to the 64 bit regions. If there are no 64 bit regions the architecture specific regions will be used.
- /SysNative  
Any access will be redirected to the architecture specific regions

In addition to these *opsi-winst* functions, we copy at the installation of the opsi-client agent the (64 bit) file `c:\windows\system32\cmd.exe` to `c:\windows\cmd64.exe`. Using this `cmd64.exe` with *Exec With* sections you may call any 64 bit operations on the command line.

Examples:

File handling:

```

if $INST_SystemType$ = "64 Bit System"
    comment ""
    comment "-----"
    comment "Testing: "
    message "64 Bit redirection"
    Files_copy_test_to_system32
    if FileExists("%System%\dummy.txt")
        comment "passed"
    else
        LogWarning "failed"
        set $TestResult$ = "not o.k."
    endif
    ExecWith_remove_test_from_system32 'cmd.exe' /C
    Files_copy_test_to_system32 /64Bit
    if FileExists64("%System%\dummy.txt")

```

```

        comment "passed"
    else
        LogWarning "failed"
        set $TestResult$ = "not o.k."
    endif
    ExecWith_remove_test_from_system32 '%SystemRoot%\cmd64.exe' /C
endif

```

Registry Handling:

```

message "Write to 64 Bit Registry"
if ($INST_SystemType$ = "64 Bit System")
    set $ConstTest$ = ""
    set $regWriteValue$ = "64"
    set $CompValue$ = $regWriteValue$
    Registry_opsi_org_test /64Bit
    ExecWith_opsi_org_test "%systemroot%\cmd64.exe" /c
    set $ConstTest$ = GetRegistryStringValue64("[HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org\test]
bitByWinst")
    if ($ConstTest$ = $CompValue$)
        comment "passed"
    else
        set $TestResult$ = "not o.k."
        comment "failed"
    endif
    set $ConstTest$ = GetRegistryStringValue64("[HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org\test]
bitByReg")
    if ($ConstTest$ = $CompValue$)
        comment "passed"
    else
        set $TestResult$ = "not o.k."
        comment "failed"
    endif
    set $regWriteValue$ = "32"
    set $CompValue$ = $regWriteValue$
    Registry_opsi_org_test
    ExecWith_opsi_org_test "cmd.exe" /c
    set $ConstTest$ = GetRegistryStringValue("[HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org\test]
bitByWinst")
    if ($ConstTest$ = $CompValue$)
        comment "passed"
    else
        set $TestResult$ = "not o.k."
        comment "failed"
    endif
    set $ConstTest$ = GetRegistryStringValue("[HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org\test]
bitByReg")
    if ($ConstTest$ = $CompValue$)
        comment "passed"
    else
        set $TestResult$ = "not o.k."
        comment "failed"
    endif
endif
else
    set $regWriteValue$ = "32"
    set $CompValue$ = $regWriteValue$
    Registry_opsi_org_test /64Bit

```

```
ExecWith_opsi_org_test "cmd.exe" /c
set $ConstTest$ = GetRegistryStringValue64("[HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org\test]
bitByWinst")
if ($ConstTest$ = $CompValue$)
    comment "passed"
else
    set $TestResult$ = "not o.k."
    comment "failed"
endif
set $ConstTest$ = GetRegistryStringValue64("[HKEY_LOCAL_MACHINE\SOFTWARE\opsi.org\test]
bitByReg")
if ($ConstTest$ = $CompValue$)
    comment "passed"
else
    set $TestResult$ = "not o.k."
    comment "failed"
endif
endif

if ($INST_SystemType$ = "64 Bit System")
    set $regWriteValue$ = "64"
    Registry_hkcu_opsi_org_test /AllNtUserDats /64Bit
    set $regWriteValue$ = "32"
    Registry_hkcu_opsi_org_test /AllNtUserDats
else
    set $regWriteValue$ = "32"
    Registry_hkcu_opsi_org_test /AllNtUserDats
    Registry_hkcu_opsi_org_test /AllNtUserDats /64Bit
endif
```

# Chapter 10

## Cook Book

This chapter contains a growing collection of examples showing real world problems that can be mastered by simple or sophisticated pieces *opsi-winst* scripting.

### 10.1 9.1. Delete a File in all Subdirectories

Since *opsi-winst* 4.2 there is an easy solution for this task: To remove a file alt.txt from all subdirectories of the user profile directory the following Files call can be used:

```
files_delete_Alt /allNtUserProfiles

[files_delete_Alt]
delete "%UserProfile%\alt.txt"
```

Nevertheless we document a workaround which could be used in older *opsi-winst* versions. It demonstrates some techniques which may be helpful for other purposes.

The following ingredients are needed:

- A DosInAnIcon section which produces a list of all directory names.
- A Files section which deletes the file alt.txt in some directory.
- A String list processing that puts the parts together.

The complete script should look like:

```
[Actions]

; variable for file name
DefVar $deleteFile$
set $deleteFile$ = "alt.txt"

; String list declarations
DefStringList list0
DefStringList list1

; capture the lines produced by the dos dir command
Set list0 = getOutputStreamFromSection ('dosbatch_profiledir')

; Loop through the lines. Call a files section for each line.
```

```

for %x$ in list0 do files_delete_x

; Here are the two special sections
[dosbatch_profiledir]
@dir "%ProfileDir%" /b

[files_delete_x]
delete "%ProfileDir%\%x%\$deleteFile$"

```

## 10.2 Check if a specific service is running

If we want to check if a specific service (exemplified with "opsiclientd") is running, and, e.g., if it is not running, start it, we may use the following script.

In order to get the list of running services we launch the command

```
net start
```

in a DosBatch section, capturing its output in list0. We trim the list, and iterate through its elements, thus seeing if the specified service is in it. If not, we do something for it.

```

[Actions]
DefStringList $list0$
DefStringList $list1$
DefStringList $result$
Set $list0%=getOutputStreamFromSection('DosBatch_netcall')
Set $list1%=getSublist(2:-3, $list0%)

DefVar $myservice$
DefVar $compareS$
DefVar $splitS$
DefVar $found$
Set $found$ = "false"
set $myservice$ = "opsiclientd"

comment "====="
comment "search the list"
; for developping loglevel = 7
; setloglevel=7
; in normal use we dont want to log the looping
setloglevel = 5
for %s% in $list1$ do sub_find_myservice
setloglevel=7
comment "====="

if $found$ = "false"
    set $result$ = getOutputStreamFromSection ("dosinanicon_start_myservice")
endif

[sub_find_myservice]
set $splitS$ = takeString (1, splitStringOnWhiteSpace("%s%"))
Set $compareS$ = $splitS$ + takeString(1, splitString("%s%", $splitS$))
if $compareS$ = $myservice$
    set $found$ = "true"

```

```
endif

[dosinicon_start_mysevice]
net start "$myservice$"

[dosbatch_netcall]
@echo off
net start
```

### 10.3 Script for Installations in the Context of a Local Administrator

Sometimes it is necessary to run an installation script as an ordinary local user and not in the context of the opsi service. For example, there are installations that require a user context or use other services that are started after a user login.

MSI installations which seem to need a local user can sometimes be configured by the option *ALLUSERS=2* to proceed without such a user:

```
[Actions]
DefVar $LOG_LOCATION$
Set $LOG_LOCATION$ = "c:\tmp\myproduct.log"
winbatch_install_myproduct

[winbatch_install_myproduct]
msiexec /qb ALLUSERS=2 /l* $LOG_LOCATION$ /i %SCRIPTPATH%\files\myproduct.msi
```

In other case it is necessary to create a temporary administrative user in whose context the installation takes place. This can be done as follows:

- Create a directory *localsetup* in the product directory (i.e. in *install\productId*).
- Move all installation files into this directory.
- Rename the installation script from *<productname>.ins* to *local\_<productname>.ins*
- Create a new *<produktname>.ins* in *install\productId* and write the statements as below documented (with variables values adapted to your situation) into it .
- Make sure that the script that is now named *local\_<produktname>.ins* finishes with a reboot call: The last executed command in the Actions section has to be the line  
`ExitWindows /Reboot`
- Insert a call at the beginning of the script *local\_<produktname>.ins* that removes the password of the temporary local administrator:

```
[Actions]
Registry_del_autologin
;....

[Registry_del_autologin]
openkey [HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon]
set "DefaultUserName"=""
set "DefaultPassword"=""
```

The *opsi-winst* script template temporarily generates a user context, executes an installation in it, then removes it. Before using the template the following values are to be set adequately:

- the value for the variable *\$ProductName\$*
- the value of the variable *\$ProductSize\$*
- *\$LockKeyboard\$* to "true".

The script proceeds as follows:

- It creates a local administrator *opsiSetupAdmin*;
- saves the autologon state;
- inserts *opsiSetupAdmin* as autologon user;
- copies the installation files to the client (as defined in *\$localFilePath\$*); among them the installation script that is to be executed in the local user context;
- creates a *RunOnce* entry in the registry that calls *opsi-winst* with the local script as argument;
- reboots in order to make the registry change work;
- when *opsi-winst* runs again, it calls an *ExitWindows /ImmediateLogout*, and the second scripting level begins to work;
- By autologon , *opsiSetupAdmin* is logged on without user interaction.
- Windows calls the *RunOnce* command, that is the *opsi-winst* call.
- The *opsi-winst* script should now regularly proceed. But at its end, there must be a *ExitWindows /ImmediateReboot* command. Otherwise the desktop would of the administrative user *opsiSetupAdmin* who is already logged at the moment would be accessible.
- after the reboot, the main script works again cleaning everything (writing back the old autologon state, deleting the local setup files, removing the *opsiSetupAdmin* profile)

We call the two involved *opsi-winst* scripts master script and local script . The first one runs in a system service context, the second which does the specific software installation runs in the context of a local administrator.



#### Caution

If the local script requires internal reboots then the master script must be adapted to produce them. As long as the local script is not finished the master script hands over control to the local script by an *ExitWindows /ImmediateLogout*. Of course the *RunOnce* entry has to be created for each run. Since username and password for the autologon are removed at the beginning of the local script they have to be reset each time as well.

There is no direct access from the local script to the product properties (usually via the String function *GetProductProperty*) . If there are values needed the master script must retrieve them and e.g. save them temporarily in the registry.

There may be product installations by external setup program calls which change registry entries which are saved by the master script and usually written back at the end of the installation. In this case the master script must be adapted to avoid writing back.

The local script runs with an administrator logged in. You have to lock the keyboard when testing is done. Otherwise anybody sitting at the client could stop script execution and take over the session.

In the following example, the password of the tempory *opsiSetupAdmin* user is set via the function *RandomStr*, which is strongly recommended.

In order to avoid logging of passwords the *loglevel* is temporarily set to -2.

**Important**

Please do not use the script as printed below, but use the opsi product: opsi-template-with-admin.

```

; Copyright (c) uib gmbh (www.uib.de)
; This sourcecode is owned by uib
; and published under the Terms of the General Public License.

; TEMPLATE for
; Skript fuer Installationen im Kontext eines temporaeren lokalen Administrators
; installations as temporary local admin
; see winst_manual.pdf / winst_handbuch.pdf

; !!! requires winst32.exe version 4.2.x !!!

;
; !!! Das lokale Installations-Skript, das durch den temporaeren lokalen Admin ausgefuehrt wird
; !!! (sein Name steht in $LocalSetupScript$), muss mit dem Befehl
; !!! exitWindows /Reboot
; !!! enden
;
;
; !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
; Vorarbeiten/Voraussetzungen/Doku pruefen wie in Winsthandbuch
; 8.3 Skript fuer Installationen im Kontext eines lokalen Administrators
; !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

[Actions]
requiredWinstVersion >= 4.10.6
setLogLevel=7
DefVar $ProductName$
DefVar $ProductSizeMB$
DefVar $LocalSetupScript$
DefVar $LockKeyboard$
DefVar $OpsiAdminPass$
DefVar $OS$
DefVar $MinorOS$
DefVar $MsVersion$
DefVar $RebootFlag$
DefVar $ShutdownRequested$
DefVar $WinstRegKey$
DefVar $RebootRegVar$
DefVar $AutoName$
DefVar $AutoPass$
DefVar $AutoDom$
DefVar $AutoLogon$
DefVar $AutoBackupKey$
DefVar $LocalFilesPath$
DefVar $LocalWinst$
DefVar $SystemType$
DefVar $platform_cmdexe$
DefVar $DefaultLogLevel$
DefVar $PasswdLogLevel$
DefVar $AdminGroup$

```

```

DefVar $SearchResult$

DefStringlist $ResultList$
DefStringlist $ResultList2$
DefStringlist $ResultList3$

comment "set $LockKeyboard$ to true to prevent user hacks while admin is logged in"
Set $LockKeyboard$="true"
Set $LockKeyboard$="false"
Set $ProductName$ = "opsi-template-with-admin"
Set $ProductSizeMB$ = "1"

Set $LocalSetupScript$ = "local_setup.ins"
set $OS$ = GetOS
set $MinorOS$ = GetNTVersion
Set $SystemType$ = GetSystemType
Set $MsVersion$ = GetMsVersionInfo
set $DefaultLogLevel$ = "7"
comment " set $PasswdLogLevel$ to 0 for production"
Set $PasswdLogLevel$="7"
SetLogLevel=$DefaultLogLevel$

if not (fileExists ("%Scriptpath%\psgetsid.exe"))
    LogError "psgetsid.exe is missing. Please install it from http://download.sysinternals.com/Files/PsTools.zip to %Scriptpath%"
    isFatalError
endif

if not(($SystemType$ = "64 Bit System") or ($SystemType$ = "x86 System"))
    LogError "Unknown Systemtype: "+$SystemType$+" - aborting"
    isFatalError
endif

if $SystemType$ = "64 Bit System"
    set $platform_cmdexe$ = "%SystemRoot%\cmd64.exe"
else
    set $platform_cmdexe$ = "%System%\cmd.exe"
endif

comment "handle Rebootflag"
Set $WinstRegKey$ = "HKLM\SOFTWARE\opsi.org\winst"
Set $RebootFlag$ = GetRegistryStringValue(["+$WinstRegKey$+" "RebootFlag")
Set $ShutdownRequested$ = GetRegistryStringValue(["+$WinstRegKey$+" "ShutdownRequested")

sub_test_autologon_data

comment "some paths required"
Set $AutoBackupKey$ = $WinstRegKey$+"\AutoLogonBackup"
Set $LocalFilePath$ = "C:\opsi_local_inst"
Set $LocalWinst$ = "c:\opsi\utils\winst32.exe"
if not( FileExists($LocalWinst$) )
    Set $LocalWinst$ = "%ProgramFilesDir%\opsi.org\preloginloader\utils\winst32.exe"
endif
if not( FileExists($LocalWinst$) )
    Set $LocalWinst$ = "%ProgramFilesDir%\opsi.org\preloginloader\opsi-winst\winst32.exe"
endif
if not( FileExists($LocalWinst$) )

```

```

        Set $LocalWinst$ = "%ProgramFilesDir%\opsi.org\opsi-client-agent\opsi-winst\winst32.exe"
    endif
    if not( FileExists($LocalWinst$) )
        LogError "No opsi-winst found. Aborting."
        isFatalError
    endif

    comment "show product picture"
    ShowBitmap /3 "%scriptpath%\localsetup\"+$ProductName$+".png" $ProductName$

    if not (($RebootFlag$ = "1") or ($RebootFlag$ = "2") or ($RebootFlag$ = "3"))
        comment "Part before first Reboot"
        comment "just reboot - this must be done if this is the first product after OS
        installation"
        comment "handle Rebootflag"
        Set $RebootFlag$ = "1"
        Registry_SaveRebootFlag
        ExitWindows /ImmediateReboot
    endif ; Rebootflag = not (1 or 2 or 3)

    if $RebootFlag$ = "1"
        comment "Part before second Reboot"

        if not(HasMinimumSpace ("%SYSTEMDRIVE%", "+$ProductSizeMB$" MB))
            LogError "Not enough space on drive C: . "+$ProductSizeMB$" MB on C: required
            for "+$ProductName$
            isFatalError
        endif

        comment "Lets work..."
        Message "Preparing "+$ProductName$" install step 1..."
        sub_Prepare_AutoLogon

        comment "we need to reboot now to be sure that the autologon work"
        comment "handle Rebootflag"
        Set $RebootFlag$ = "2"
        Registry_SaveRebootFlag
        sub_test_autologon_data
        ExitWindows /ImmediateReboot
    endif ; Rebootflag = not (1 or 2)

    if ($RebootFlag$ = "2")
        comment "Part after first Reboot"

        comment "handle Rebootflag"
        Set $RebootFlag$ = "3"
        Registry_SaveRebootFlag

        comment "Lets work..."
        Message "Preparing "+$ProductName$" install step 2..."
        Registry_enable_keyboard /64bit
        comment "now let the autologon work"
        comment "it will stop with a reboot"
        ExitWindows /ImmediateLogout
    endif ; Rebootflag = 2

    if ($RebootFlag$ = "3")

```

```

    comment "Part after second Reboot"

    comment "handle Rebootflag"
    Set $RebootFlag$ = "0"
    Registry_SaveRebootFlag

    comment "Lets work..."
    Message "Cleanup "+$ProductName$+" install (step 3)..."
    sub_Restore_AutoLogon
    set $SearchResult$ = GetRegistryStringValue64("[HKLM\SOFTWARE\Microsoft\Windows\
CurrentVersion\RunOnce] opsi_autologon_setup")
    if $SearchResult$ = $LocalWinst$+" "+$LocalFilePath$+"\ "+$LocalSetupScript$+" /batch"
        LogError "Localscript did not run. We remove the RunOnce entry and abort"
        Registry_del_runonce /64Bit
        isFatalError
    endif
    comment "This is the clean end of the installation"
endif ; Rebootflag = 3
ExitWindows /Reboot

[sub_Prepare_AutoLogon]
comment "copy the setup script and files"
Files_copy_Setup_files_local
comment "read actual AutoLogon values for backup"
set $AutoName$ = GetRegistryStringValue64("[HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
Winlogon] DefaultUserName")
comment "if AutoLogonName is our setup admin user, something bad happend"
comment "then let us cleanup"
if ($AutoName$="opsiSetupAdmin")
    set $AutoName$=""
    set $AutoPass$=""
    set $AutoDom$=""
    set $AutoLogon$="0"
else
    set $AutoPass$ = GetRegistryStringValue64("[HKLM\SOFTWARE\Microsoft\Windows NT\
CurrentVersion\Winlogon] DefaultPassword")
    set $AutoDom$ = GetRegistryStringValue64("[HKLM\SOFTWARE\Microsoft\Windows NT\
CurrentVersion\Winlogon] DefaultDomainName")
    set $AutoLogon$ = GetRegistryStringValue64("[HKLM\SOFTWARE\Microsoft\Windows NT\
CurrentVersion\Winlogon] AutoAdminLogon")
endif

comment "backup AutoLogon values"
Registry_save_autologon

comment "prepare the admin AutoLogon"
SetLogLevel=$PasswdLogLevel$
set $OpsAdminPass$= randomstr
Registry_autologon /64Bit

comment "get the name of the admin group"
comment "using psgetsid from sysinternals pstools"
set $ResultList$ = getOutStreamFromSection("DosInAnIcon_get_admin_group")
set $AdminGroup$ = takeString(6,$ResultList$)
set $AdminGroup$ = takeString(1,splitstring($AdminGroup$,"\"))
comment "create our setup admin user"
DosInAnIcon_makeadmin

```

```
SetLogLevel=$DefaultLogLevel$

comment "remove c:\tmp\winst.bat with password"
Files_remove_winst.bat

comment "store our setup script as run once"
Registry_runOnce /64Bit

comment "disable keyboard and mouse while the autologin admin works"
if ($LockKeyboard$="true")
    Registry_disable_keyboard /64Bit
endif

[sub_test_autologon_data]
set $AutoPass$ = GetRegistryStringValue64(" [HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
Winlogon] DefaultPassword")
set $AutoDom$ = GetRegistryStringValue64(" [HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
Winlogon] DefaultDomainName")
set $AutoLogon$ = GetRegistryStringValue64(" [HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\
Winlogon] AutoAdminLogon")

[sub_Restore_AutoLogon]
comment "read AutoLogon values from backup"
set $AutoName$ = GetRegistryStringValue("["+ $AutoBackupKey$+"] DefaultUserName")
set $AutoPass$ = GetRegistryStringValue("["+ $AutoBackupKey$+"] DefaultPassword")
set $AutoDom$ = GetRegistryStringValue("["+ $AutoBackupKey$+"] DefaultDomainName")
set $AutoLogon$ = GetRegistryStringValue("["+ $AutoBackupKey$+"] AutoAdminLogon")

comment "restore the values"
SetLogLevel=$PasswdLogLevel$
Registry_restore_autologon /64Bit
SetLogLevel=$DefaultLogLevel$
comment "delete our setup admin user"
DosInAnIcon_deleteadmin
comment "cleanup setup script, files and profiledir"
Files_delete_Setup_files_local
comment "delete profiledir"
DosInAnIcon_deleteprofile

[Registry_save_autologon]
openkey [ $AutoBackupKey$]
set "DefaultUserName"=$AutoName$
set "DefaultPassword"=$AutoPass$
set "DefaultDomainName"=$AutoDom$
set "AutoAdminLogon"=$AutoLogon$

[Registry_restore_autologon]
openkey [HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon]
set "DefaultUserName"=$AutoName$
set "DefaultPassword"=$AutoPass$
set "DefaultDomainName"=$AutoDom$
set "AutoAdminLogon"=$AutoLogon$

[DosInAnIcon_deleteadmin]
NET USER opsiSetupAdmin /DELETE

[Registry_SaveRebootFlag]
```

```

openKey [%WinstRegKey$]
set "RebootFlag" = "$RebootFlag$"

[Files_copy_Setup_files_local]
copy -s %ScriptPath%\localsetup\*.* $LocalFilesPath$

[Files_delete_Setup_files_local]
delete -sf $LocalFilesPath$
; folgender Befehl funktioniert nicht vollständig, deshalb ist er zur Zeit auskommentier
; der Befehl wird durch die Sektion "DosInAnIcon_deleteprofile" ersetzt (P.Ohler)
;delete -sf "%ProfileDir%\opsiSetupAdmin"

[DosInAnIcon_deleteprofile]
rmdir /S /Q "%ProfileDir%\opsiSetupAdmin"

[DosInAnIcon_get_admin_group]
@echo off
"%ScriptPath%\psgetsid.exe" /accepteula S-1-5-32-544

[DosInAnIcon_makeadmin]
NET USER opsiSetupAdmin $OpsAdminPass$ /ADD
NET LOCALGROUP $AdminGroup$ /ADD opsiSetupAdmin

[Registry_autologon]
openkey [HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon]
set "DefaultUserName"="opsiSetupAdmin"
set "DefaultPassword"="$OpsAdminPass$"
set "DefaultDomainName"="localhost"
set "AutoAdminLogon"="1"

[Registry_runonce]
openkey [HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce]
set "opsi_autologon_setup"=' "$LocalWinst$" "$LocalFilesPath$\LocalSetupScript$" /batch'

[Registry_del_runonce]
openkey [HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce]
DeleteVar "opsi_autologon_setup"

[Registry_disable_keyboard]
openkey [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Kbdclass]
set "Start"=REG_DWORD:0x4
openkey [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Mouclass]
set "Start"=REG_DWORD:0x4

[Registry_enable_keyboard]
openkey [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Kbdclass]
set "Start"=REG_DWORD:0x1
openkey [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Mouclass]
set "Start"=REG_DWORD:0x1

[Files_remove_winst_bat]
delete -f c:\tmp\_winst.bat

```

## 10.4 XML File Patching: Setting Template Path for OpenOffice.org 2

Setting the template path can be done by the following script extracts

```
[Actions]
; ....

DefVar $oooTemplateDirectory$
;-----
;set path here:

Set $oooTemplateDirectory$ = "file://server/share/verzeichnis"
;-----
;...

DefVar $sofficePath$
Set $sofficePath$= GetRegistryStringValue (" [HKEY_LOCAL_MACHINE\SOFTWARE\OpenOffice.org\
    OpenOffice.org\2.0] Path")
DefVar $oooDirectory$
Set $oooDirectory$= SubstringBefore ($sofficePath$, "\program\soffice.exe")
DefVar $oooShareDirectory$
Set $oooShareDirectory$ = $oooDirectory$ + "\share"

XMLPatch_paths_xcu $oooShareDirectory$+"\registry\data\org\openoffice\Office\Paths.xcu"
; ...

[XMLPatch_paths_xcu]
OpenNodeSet
- error_when_no_node_existing false
- warning_when_no_node_existing true
- error_when_nodecount_greater_1 false
- warning_when_nodecount_greater_1 true
- create_when_node_not_existing true
- attributes_strict false

documentroot
all_childelements_with:
elementname: "node"
attribute:"oor:name" value="Paths"
all_childelements_with:
elementname: "node"
attribute: "oor:name" value="Template"
all_childelements_with:
elementname: "node"
attribute: "oor:name" value="InternalPaths"
all_childelements_with:
elementname: "node"

end

SetAttribute "oor:name" value="$oooTemplateDirectory$"
```

## 10.5 Retrieving Values From a XML File

As treated in Section 10.4, *opsi-winst* can evaluate and modify XML files.

An example shall demonstrate how a value can be retrieved from a XML file. We assume that the following XML file is:

```
<?xml version="1.0" encoding="utf-16" ?>
<Collector xmlns="http://schemas.microsoft.com/appx/2004/04/Collector" xmlns:xs="http://www.w3.
  org/2001/XMLSchema-instance" xs:schemaLocation="Collector.xsd" UtcDate="04/06/2006 12:28:17"
  LogId="{693B0A32-76A2-4FA0-979C-611DEE852C2C}" Version="4.1.3790.1641" >
  <Options>
    <Department></Department>
    <IniPath></IniPath>
    <CustomValues>
      </CustomValues>
    </Options>
  <SystemList>
    <ChassisInfo Vendor="Chassis Manufacture" AssetTag="System Enclosure 0" SerialNumber="EVAL"
    />
    <DirectxInfo Major="9" Minor="0"/>
  </SystemList>
  <SoftwareList>
    <Application Name="Windows XP-Hotfix - KB873333" ComponentType="Hotfix" EvidenceId="256"
    RootDirPath="C:\WINDOWS\NtUninstallKB873333$\spuninst" OsComponent="true" Vendor="Microsoft
    Corporation" Crc32="0x4235b909">
      <Evidence>
        <AddRemoveProgram DisplayName="Windows XP-Hotfix - KB873333" CompanyName="Microsoft
        Corporation" Path="C:\WINDOWS\NtUninstallKB873333$\spuninst" RegistryPath="
        HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall\KB873333"
        UninstallString="C:\WINDOWS\NtUninstallKB873333$\spuninst\spuninst.exe" OsComponent="true"
        UniqueId="256"/>
      </Evidence>
    </Application>
    <Application Name="Windows XP-Hotfix - KB873339" ComponentType="Hotfix" EvidenceId="257"
    RootDirPath="C:\WINDOWS\NtUninstallKB873339$\spuninst" OsComponent="true" Vendor="Microsoft
    Corporation" Crc32="0x9c550c9c">
      <Evidence>
        <AddRemoveProgram DisplayName="Windows XP-Hotfix - KB873339" CompanyName="Microsoft
        Corporation" Path="C:\WINDOWS\NtUninstallKB873339$\spuninst" RegistryPath="
        HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall\KB873339"
        UninstallString="C:\WINDOWS\NtUninstallKB873339$\spuninst\spuninst.exe" OsComponent="true"
        UniqueId="257"/>
      </Evidence>
    </Application>
  </SoftwareList>
</Collector>
```

To read the elements and get the values of all „Application“ nodes we may use these extracts of code:

```
[Actions]
DefStringList $list$

...

set $list$ = getReturnListFromSection ('XMLPatch_findProducts '+$TEMP$+'\test.xml')
for $line$ in $list$ do Sub_doSomething
```

```
[XMLPatch_findProducts]
openNodeSet
    ; Node „Collector“ is documentroot
    documentroot
    all_childelements_with:
        elementname:"SoftwareList"
    all_childelements_with:
        elementname:"Application"
end
return elements

[Sub_doSomething]
set $escLine$ = EscapeString:$line$
; now we can work on the content of $escLine$
```

We encapsulate the retrieved Strings by setting their values as a whole into an variable via an EscapeString call. Since the loop variable %line% is not a common variable but behaves like a constant all special characters in it (as < > \$ % “ ’) may cause difficulties.

## 10.6 Inserting a Name Space Definition Into a XML File

The *opsi-winst* XMLPatch section requires fully declared XML name spaces (as is postulated in the XML RFC). But there are XML configuration files which do not declare „obvious“ elements (and the interpreting programs insist that the file looks this way). Especially patching the lots of XML/XCU configuration files of OpenOffice.org proved to be a hard job. For solving this task, A. Pohl (many thanks!) the functions XMLaddNamespace and XMLremoveNamespace. Its usage is demonstrated by the following example:

```
DefVar $XMLFile$
DefVar $XMLElement$
DefVar $XMLNameSpace$
set $XMLFile$ = "D:\Entwicklung\OPSI\winst\Common.xcu3"
set $XMLElement$ = 'oor:component-data'
set $XMLNameSpace$ = 'xmlns:xml="http://www.w3.org/XML/1998/namespace"'

if XMLaddNamespace($XMLFile$, $XMLElement$, $XMLNameSpace$)
    set $NSMustRemove$="1"
endif
;
; now the XML Patch should work
; (commented out since not integrated in this example)
;
; XMLPatch_Common $XMLFile$
;
; when finished we rebuild the original format
if $NSMustRemove$="1"
    if not (XMLremoveNamespace($XMLFile$, $XMLElement$, $XMLNameSpace$))
        LogError "XML-Datei konnte nicht korrekt wiederhergestellt werden"
        isFatalError
    endif
endif
```

Please observe that the XML file must be formatted such that the element tags do not contain line breaks.

## Chapter 11

# Special Error Messages

- No Connection with the opsi Service  
The *opsi-winst* logs: "... cannot connect to service".

The information which is shown additionally may give a hint to the problem:

**Socket-Fehler #10061, Connection refused**

Perhaps the opsi service does not run.

**Socket-Fehler #10065, No route to host**

No network connection to server

**HTTP/1.1. 401 Unauthorized**

The service responds but the user/password combination is not accepted.