PC Software Installation using wlnst

- the opsi Windows Installer -

Manual for software developers

Revision date: 25/05/09

uib gmbh (www.uib.de)

Content

1 Windows Installer 8

2 Command Line Parameters 9

3 Additional Configurations 12

3.1 Central Logging of Error Messages 12

3.2 Skinnable wInst 13

4 The wInst Script 14

4.1 An Example 14

4.2 Primary and Secondary Subprograms of a wInst script 16

4.3 String Expressions in a wInst Script 17

5 Definition and Use of Variables and Constants in a wInst Script 19

5.1 Overview 19

5.2 Global Text Constants 20

5.2.1 Usage 20

5.2.2 Example 20

5.2.3 List of Existing Constants 20

(i) System Paths 21

(ii) wInst Paths 22

(iii) Network Information 22

(iv) Data for opsi service 23

5.3 String (or Text) Variables 23

5.3.1 Declaration 23

5.3.2 Value Assignment 24

5.3.3 Use of variables in String expressions 25

5.3.4 Secondary vs. primary sections 25

5.4 Stringlist Variables 26

6 Syntax and Meaning of Primary Sections of a wInst Script 27

6.1 Primary Sections 27

6.2 Parametrizing wInst 28

6.2.1 Example 28

6.2.2 Specification of Logging Level 28

6.2.3 Required wInst Version 29

6.2.4 Reacting on Errors ... 29
6.2.5 Staying On Top ... 30
6.3 String Expressions, String Values, and String Functions ... 31
6.3.1 Elementary String Values ... 31
6.3.2 Strings in Strings (Nested String Values) ... 31
6.3.3 String Concatenation ... 32
6.3.4 String Variables ... 32
6.3.5 String Functions which Return the OS Type ... 32
6.3.6 String Functions for Retrieving Environment or Command Line Parameters ... 33
6.3.7 Reading Values from the Windows Registry and Transforming Values into Registry Format ... 34
6.3.8 Reading Property Values ... 34
6.3.9 Retrieving Data from etc/hosts ... 36
6.3.10 String processing ... 36
6.3.11 Additional String Functions ... 38
6.3.12 (String-) functions for licence management ... 38
6.4 String List Functions and String List Processing ... 39
6.4.1 Info Maps ... 39
6.4.2 Producing String Lists from Strings ... 42
6.4.3 Loading the Lines of a Text File into a String List ... 42
6.4.4 Simple String Values generated from String Lists ... 43
6.4.5 Producing String Lists from wInst Sections ... 44
6.4.6 Transforming String Lists ... 45
6.4.7 Iterating through String Lists ... 45
6.5 Special Commands ... 46
6.6 Commands for User Information and User Interaction ... 47
6.7 Conditional Statements (if Statements) ... 49
6.7.1 Example ... 49
6.7.2 General Syntax ... 49
6.7.3 Boolean Expressions ... 50
6.8 Subprogram Calls ... 52
6.8.1 Syntax of Procedure Calling ... 53
6.9 Controlling Reboot ... 54
6.10 Keeping Track of Failed Installations ... 57

7 Secondary Sections ... 59

7.1 Files Sections ... 59
7.1.1 Example ... 59
7.1.2 Call Parameters ... 60
7.1.3 Commands ... 61
7.2 Patches-Sektionen ... 63
7.2.1 Example ... 63

7.2.2 Call Parameter....64
7.2.3 Commands....64
7.3 PatchHosts Sections....65
7.4 IdapiConfig Sections....67
7.5 PatchTextFile Sections....67
7.5.1 Example....68
7.5.2 Call Parameter....68
7.5.3 Commands....68
7.6 LinkFolder Sections....70
7.6.1 Windows....70
7.6.2 Linux....72
7.7 XMLPatch Sections....76
7.7.1 Structure of a XML Document....77
7.7.2 Options for Selection a Set of Elements....79
(i) Explicit Syntax....79
(ii) Short Syntax....79
(iii) Selecting by Textual Content (only for explicit syntax)....80
(iv) Parametrizing Search Strategy....80
7.7.3 Patch Actions....81
7.7.4 Returning Lists to the Caller....82
7.8 ProgmanGroups Sections....83
7.9 WinBatch Sections....83
7.10 DOSBatch/ShellBatch Sections....84
7.10.1 Windows....84
7.10.2 Linux....85
7.11 DOSInAnIcon/ShellInAnIcon Sections....85
7.11.1 Windows....85
7.11.2 Linux....85
7.12 Registry Sections....86
7.12.1 Example....86
7.12.2 Call Parameters....86
7.12.3 Commands....87
7.12.4 Registry Sections to Patch "All NTUser.dat"....90
7.12.5 Registry Sections in Regedit Format....91
7.12.6 Registry Sections in AddReg Format....92
7.13 OpsiServiceCall Sections....92
7.13.1 Call Parameters....93
7.13.2 Section Format....94
7.14 ExecPython Sections....95
7.14.1 Example....95
7.14.2 Interweaving a Python Script with the wInst Script....96

7.15 ExecWith Sections....97
7.15.1 Call Syntax....97
7.15.2 More Examples....98
8 Cook Book....99
8.1 Delete a File in all Subdirectories....99
8.2 Check if a Specific Service is Running....100
8.3 Script for Installations in the Context of a Local Administrator....101
8.4 XML File Patching: Setting Template Path for OpenOffice.org 2....108
8.5 Retrieving Values From a XML File....109
8.6 Inserting a Name Space Definition Into a XML File....111
9 No Connection with the opsi Service....113

Revison history of this manual

wInst version 4.8.6 (opsi version 3.4)

New Boolean function opsiLicenseManagementEnabled (cf. section 6.7.3)

New String functions DemandLicenseKey, FreeLicenseKey (section 6.3.12),

getLastServiceErrorClass, getLastServiceErrorMessage (section 6.3.13)

wInst version 4.8.4 (opsi version 3.3.1)

New version check option -V for copy actions, meaning version check only with regard to files in the target directory (cf. section 7.1.2)

wInst version 4.8.1 (opsi version 3.3.1)

New constant %installingProduct% (section 5.2.3 (iv)).

For licence management: new String functions demandLicenseKey, freeLicenseKey

wInst version 4.7.4 (opsi version 3.3.1)

New OS version functions GetMSVersionInfo (major + minor version info as given by the WinApi)

GetSystemType (for XP and Vista, possible values '64 Bit System' or 'x86 System'

wInst version 4.6.0 (opsi version 3.3)

wInst got a new skin which is editable (cf. section 3.2)

wInst version 4.5.9 (opsi version 3.2 updated)

New StringList functions getLocaleInfoMap and getFileVersionMap (section 6.4.1)

New String function getValue($key, $map) for a String $key and Stringlist $map (section 6.4.4)

New copy modifier -c (cf. section 7.1.3)

New constants %ipAddress%, %ipName% (cf. section 5.2.3).

New String function getLastExitCode. It returns the ExitCode – or ErrorLevel – of the last winbatch call. (sections6.3.6)

New String function trim.

New commands for primary sections: sleepSeconds, markTime, diffTime (cf. section 6.6)

New section type ExecWith (cf. section 7.15)

wInst version 4.5.6 (opsi version 3.2 updated)

New variant of the ExitWindows command (/ShutdownWanted, cf. section 6.9).

wInst version 4.5 (packed with opsi version 3.2)

New section type execPython (section 7.14). If python is installed on the system, python.exe is called and the section interpreted as a python script. For interweaving the python script with the winst script there are new constants %opsiserviceURL%, %opsiserviceUser%, %opsiservicePassword%, %hostID%, %logfile% (cf. 5.2.3) and a new String function getLoglevel (shortly loglevel; cf. 6.3.11).

wInst version 4.4 (packed with opsi version 3.1)

New section type opsiServiceCall (section 7.13) for connecting directly - or with an interactively supplied password - to and communicating with an opsi service.

New functions XMLaddNamespace and XMLRemoveNamespace (cf. section 6.7.3 and cookbook 8.6)

wInst version 4.3 (required for opsi version 3.0)

New appendix (section 9.1) on error messages in the situation that the connect to the opsi service fails.

Corrected description for the *WaitForProcessEnding* option for the winbatch section.

The opsi service (opsi Version 3.0) can inform on the PC configuration (Section 2 of this manual)

By the new function *requiredWinstVersion* (cf. section 6.3.3) a **wInst** script can check if the installed **wInst** meets its requirements.

wInst version 4.2 (packed with opsi version 2.5)

Supports the state description "failed" (section 6.10)

New RandomStr function (cf. sections 6.2.9, 8.3)

Pseudo function EscapeString (section 6.3.2)

For Files sections with Option /allNtUserProfiles the new variable %UserProfileDir% can be used (section 7.1.2)

wInst constants can now be used in *sub* sections (section 6.1)

A new LogLevel syntax can be used (section 6.1.2)

wInst version 4.1

New parameter /WaitForProcessEnding for *WinBatch* calls (section 7.9)
Parameter /ImmediateLogout for ExitWindows-Kommando eingefuehrt (section 6.9, 8.3)

Syntaxvariante /regedit fuer Registry-Sektionen (section 7.12)

New string list function loadUnicodeTextFile (section 6.4.1, 7.12.4)

A sub section can be called with a string list expression as parameter (section 6.8.1)

wInst version 4.0

Introduces a kind of string list processing (sections 5.4, 6.4, 8.2 ,...)

Capturing of the output of DosBatch/Shell calls as string lists (section 6.4.4)

Patches of XML files (section 7.7)

1 Windows Installer

The open source program **wInst** (or **windows Installer**) serves in the context of ***opsi*** – open pc server integration (cf. 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.

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 **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.):

- **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 occured 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.

***wInst** in dialog and test mode*

2 Command Line Parameters

wInst kann be started with different sets of parameters depending on context and purpose of use.

There are the following syntactical schemata:

(1) Show usage:

wInst /?

wInst /h[elp]

(2) Execute a script:

wInst ***scriptfile*** **[** ***[/logfile] logfile]***
[/batch **|** ***/ini winstconfigfilepath]***
[/parameter parameterstring]

(3) Read the PC configuration from the opsi service and act accordingly, since **wInst 4.3**

wInst ***/opsiservice [opsiserviceurl]***
[/clientid clientname]
[/username username]
[/password password]
[[/logfile] logfile]
[/parameter parameterstring]]

(4) Read the PC profile file and act accordingly (opsi classic)

wInst /pcprofil
[*PC_configuration_file* [[/logfile] ***logfile*****]]**
[/parameter ***parameterstring*****]**

In each case we have:

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

Explanations to (2):

- If option **/batch** is used, then **wInst** shows only its "batch surface" offering no user dialogs. 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 ***/ini*** refers to a file in ini file format that holds the last used (in interactive mode) script file names. The dialog 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 (3):

- If a **opsiserviceurl** is missing the following URL is used:

  https://DEPOTSERVER:4447

  where **DEPOTSERVER** is the server name derived from the value of depoturl in the Windows Registry.
- Default value for **clientid** is the computer name.

Explanations to (4):

- In opsi classic, **wInst** reads the PC specific data directly from the PC configuration file - the so called PC profile file or "ini file" since it has ini file format. If an explicit file name is missing the "classic" default **P:\PCPatch\%PCNAME%.ini** is used where **%PCNAME%** is an appropriately set environment variable.
- In particular, the PC configuration file informs which applications shall be installed. The pathes of the **wInst** scripts that control the installations are read from the file **pathnams.ini** that has as default location **p:\pcpatch** steht.
- The not interactive mode is implied.

It is possible to overwrite the log file location by data from PC configuration file or by the opsi service.

To do this we must a create a section in the PC configuration file that looks like (if the log file shall be placed in the directory n:\tmp with file name xxx.log):

- [winst]

  Logdateiname=n:\tmp\xxx.log

3 Additional Configurations

3.1 Central Logging of Error Messages

If wanted, **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 1, **wInst** tries to open a $pcname$.err in the configshare, subdirectory pcpatch\pclog, and write the data to it.
- 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:

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

3.2 Skinnable **wInst**

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

4 The wInst Script

On principle: **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 **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 devoleped.

4.1 An Example

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 **wInst** script looks like this one (here with a check which operating system is installed):

```
[Initial]
Message "Installation of Mozilla"
   LogLevel=2

[Aktionen]
;Determine the OS
DefVar $0S$
Set $OS$ = GetOS
; Windows NT family (including Win2k, WinXP)
; or Win95 (including Win98, WinME)
; or Linux

;Which NT-Version?
DefVar $NTVersion$

if $OS$ = "Windows_95"
  Sub_install_win95

else
  Set $NTVersion$ = GetNTVersion
```

```
  ; has values "NT4" or "Win2k"  or "WinXP"
  ; or "Win NT " + majorVersion + "." + minorVersion

  if ( $NTVersion$ = "NT4" ) or ( $NTVersion$ = "Win2k" )
    sub_install_winnt
  else
    if ( $NTVersion$ = "WinXP" )
          sub_install_winXP
    else
          stop "OS not supported"
    endif
  endif

else
  stop "OS not supported"

endif


[sub_install_win95]
Files_Kopieren_95
WinBatch_Setup

[sub_install_winNT]
Files_Kopieren_NT
WinBatch_Setup

[sub_install_winXP]
Files_Kopieren_XP
WinBatch_SetupXP

[Files_Kopieren_95]
copy "%scriptpath%\files_win95\*.*" "c:\temp\installation"

[Files_Kopieren_NT]
copy "%scriptpath%\files_winnt\*.*" "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?

4.2 Primary and Secondary Subprograms of a **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 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 **wInst** script begins with working on the sections `[Initial]` and `[Aktionen]` (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 **Aktionen** sections and may contain calls for further subroutines.

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

The *primary or general control sections* comprise

- the **Initial** section (by convention the beginning of the script),
- the **Aktionen** section (should follow to Initial section), and
- **Sub** sections (0 to n subroutines called by the **Aktionen** section which are syntactical and logical extensions of the calling section).

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** sections,
- **DosInAnIcon/ShellInAnIcon** sections,
- **Registry** sections,

- **Patches** sections,
- **PatchHosts** sections,
- **PatchTextFile** sections,
- **StartMenu** sections,
- **ProgmanGroups** sections (deprecated),
- **IdapiConfig** sections,
- **XMLPatch** sections.

Meaning and syntax of the different section types is treated in chapters 5 and 6.

4.3 String Expressions in a 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

  ```
  $NtVersion$
  ```

  may stand for "`Windows_NT`" – if it has been assigned beforedhand 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:

  ```
  GetOs
  ```

On a NT system, this function call yields the value "`Windows_NT`" (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 6.3)

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 **`wInst`** script:

5 Definition and Use of Variables and Constants in a wInst Script

5.1 Overview

In a **wInst** script, variables and constants appear as "words", that are interpreted by **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 ("+") .

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 preset by the **wInst** program and cannot be changed in a script. Before interpreting the script **wInst** *replaces* each occurence 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 **wInst** found and read the script that it just executes. This location may be, e.g., `p:\install\produkt`. Then we have to write

  "%ScriptPath%"

  in the script when we want do get the value

  `"p:\install\produkt"`

  - 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:

5.2 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 pathes. **`wInst`** introduces such values as *constants* into the script.

5.2.1 Usage

The fundamental characteristics of a text constant is the way how the values which it represents come intro 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).

5.2.2 Example

`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%"
```

5.2.3 List of Existing Constants

At this moment the following constants are implemented:

(i) System Paths

- **%AppdataDir%**

  The default value since Windows 2000 in a german context is:

  ```
  C:\Dokumente und Einstellungen\%USERNAME%\Anwendungsdaten
  ```

- **%AllUsersProfileDir%**

  Since Windows 2000:

  ```
  C:\Dokumente und Einstellungen\All Users
  ```

- **%CommonStartMenuPath%**

  Default:

  ```
  C:\Dokumente und Einstellungen\All Users\Startmenü
  ```

- **%ProfileDir%**

  Since Windows 2000:

  ```
  C:\Dokumente und Einstellungen
  ```

  Hint:

  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.

- **%ProgramFilesDir%**

  By default:

  ```
  C:\Programme
  ```

- **%Systemroot%**

  Denotes the root directory for Windows on the work station (without closing backslash) - e.g.

  ```
  c:\windows
  c:\winnt
  ```

- **%System%**

  Name of the Windows system directory (without backslash) e.g.

  ```
  c:\windows\system
  c:\winnt\system32
  ```

- **%Systemdrive%**

  Denotes the drive on which the operating system is installed.

(ii) wInst Paths

- **%ScriptPath%**

  represents the path of the current 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 wInst script is located (including the colon).

- **%WinstDir%**

  The location (without closing backslash) of the running **wInst**.

- **%Logfile%**

  The name of the logfile which **wInst** is using.

(iii) 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.

- **%IPAddress%**
  The network IP address.

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

(iv) Data for 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.

- **%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 **wInst** logging functions.

- **%installingProduct%**

  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.

5.3 String (or Text) Variables

5.3.1 Declaration

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

```
DefVar <variable name>
```

e.g.

```
DefVar $NTVersion$
```

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, **Aktionen** section and **sub** sections).
- The declaration should not depend on a condition. That is it should *not 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 ("") .

5.3.2 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 <Variablenname> = <Value>
  ```

  <Value> means any (String valued) expression.

  Examples (cf. section 6.3):

```
Set $OS$ = GetOS
Set $NTVersion$ = "not determined"

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

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

5.3.3 Use of variables in String expressions

- In primary sections of a **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.

5.3.4 Secondary vs. primary sections

When a secondary section is loaded and **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"
```

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 occurences virtually in the same moment).

5.4 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 (section 6.3).

6 Syntax and Meaning of Primary Sections of a **wInst** Script

As shortly presented in chapter 4 the `Aktionen` section of a script can be regarded as a the main method of the **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 **wInst** script.

6.1 Primary Sections

There are possibly three kinds of primary sections in a script

- an **Initial** section,
- an **Aktionen** section,
- any number of **Sub** sections

`Initial` and `Aktionen` 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 `Aktionen` section can be regarded as the main program in a **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 `Aktionen` 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 `Aktionen` 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]`

`Sub` sections of second and higher order (subs of subs and so on) can not have any more internal sections but must refer to external sections (for this distinction cf. 6.8).

6.2 Parametrizing **wInst**

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

6.2.1 Example

```
[Initial]
LogLevel=2
ExitOnError=false
ScriptErrorMessages=on
TraceMode=off
```

This means that:

- logging level is set to 2,
- when an error occurs **wInst** shall try to continue script execution,
- if a script syntax error occurs it shall be communicated (this will be in a special window), and
- we dont want to activate the trace mode for script execution (which would mean that we asked if we want to continue after each program step).

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

To the details of syntax and meaning:

6.2.2 Specification of Logging Level

There are two syntactical variants for specifying the logging level:

```
LogLevel = <number>
LogLevel = <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, **wInst** tries to evaluate the string expression as a number.

There exist six levels from -2 up to +3.

LogLevel = 0 (Error Level) has the meaning that only a summary of events is produced. Only errors and extraordinary events are logged more in detail.

With **LogLevel = 1** (Warning Level) we tell the program that we wish to receive also warnings - meaning indications of events that were possibly not intended and may lead to errors or misbehaviour.

At **LogLevel = 2** (default) every operation shall be logged.

With **Level = 3** some additional debugging information may be given.

Level = -1 reduces the logging to errors.

A possibly useful setting may be **LogLevel = -2.** *Any logging* (besides of **comment**s) is *turned off*.

6.2.3 Required wInst Version

The statement

- **requiredWinstVersion <RELATIONSSYMBOL> <ZAHLENSTRING>**

e.g.

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

makes **wInst** check if the desired version state is given. Otherwise an error message windows pops up.

This feature exists since **wInst** version 4.3. For an earlier version, the statement is unknown, and the statement itsself 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 actual used **wInst** version as long as the required version is at least version 4.3.

6.2.4 Reacting on Errors

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

1. illegal statements which cannot be interpreted by **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 analyzed later.

The behaviour of **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.

6.2.5 Staying On Top

StayOnTop = <boolean value>

With **StayOnTop = true** (or = **on**) we request, that - in *batch mode* - the **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.

Observe: According to the system manual the value cannot be changed while the proram 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 **wInst** keeps staying on top.

6.3 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

6.3.1 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>:

We have e.g.

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

6.3.2 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"'
```

6.3.3 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"**.

6.3.4 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 5.3.

The following sections present the variety of string functions.

6.3.5 String Functions which Return the OS Type

- **GetOS**
  The function tells which type of operating system is running. It returns one of the following values:
  "Windows_16"
  "Windows_95" (including Windows 98 and ME)
  "Windows_NT" (including Windows 2000 and XP)
  "Linux"

- **GetNtVersion**
  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"

- **GetSystemType**

  checks for a Windows NT System if it can be assumed that the system is 64 Bit. In this case the value is **"64 Bit System"** otherwise **"x86 System"**.

6.3.6 String Functions for Retrieving Environment or Command Line Parameters

- **EnvVar (<String expression>)**
  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 **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.

6.3.7 Reading Values from the Windows Registry and Transforming Values into Registry Format

- **GetRegistryStringValue (<String expression>)**
  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.

  The function

- **RegString (<String expression>)**
  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\\"

6.3.8 Reading Property Values

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 (<String expression> [<character sequence>] <character sequence>)**

  The **<String expression>** is interpreted as file name, the first **<character sequence>** as section name, the second as key name. I.e.,

  ```
  GetIni ("MYINIFILE" [mysection] mykey)
  ```

  returns the same value as

  ```
  GetValueFromInifile ("MYINIFILE", "mysection", "mykey", "")
  ```

  E.g.

  ```
  GetIni ("%Systemroot%\win.ini" [Interbase] RootDirectory)
  ```

  yields the entry of section **[Interbase]** of the Windows main inifile.

The third function returns a PC specific property of the product which is just being installed (**wInst** running in pcprofile mode). Its syntax reads

- **IniVar (<String Expression>)**

  E.g.

IniVar ("switch")

is in "opsi classic" – with default configuration paths and if the product just being installed is named `PRODUCT` – short for

```
GetValueFromIniFile ("p:\pcpatch\%PCNAME%.ini", "PRODUCT-install", "switch",
"")
```

If **wInst** is connected to the opsi configuration service (opsi 3.0) the product property is retrieved from the service (no matter if it is permanently saved in an ini file or somewhere else).

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

```
IniVar ("viewer")
```

resp.

```
IniVar ("policy")
```

6.3.9 Retrieving Data from etc/hosts

- **GetHostsName (<String expression>)**
  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\"`.

  Inversely, backed by the same files,

- **GetHostsAddr (<String expression>)**
  tells the IP address to a given host or alias name.

6.3.10 String processing

- **ExtractFilePath (<String expression>)**

interprets the passed String value as file or path name and returns the path part (the string up to the last "\", including it).

`StringSplit (String1, String2,index)`

is deprecated. The expression is equivalent to

```
takeString(INDEX, splitString (String1, String2)
```

(cf. the section String list processing, section 6.4).

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).

E. g. ,

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

produces the String value

```
"share"
```

For, numbering the parts of the string sliced by "\" we get

index 0: "" (empty string before the first occurrence of "\"
index 1: "" (empty string between the first and second "\")
index 2: "server"
index 3: "share"

`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 (stringValue1, stringValue2)**
  yields the sequence of characters of stringValue1 up to the beginning of stringValue2 ,

  E. g.

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

  returns

  ```
  "C:\programme\staroffice"
  ```

- **Trim(stringValue)**

  cuts leading and trailing white space from **stringValue.**

6.3.11 Additional String Functions

- **RandomStr**
  returns a random String of length 10 where upper case letters, lower case letters and digits are mixed (for creating passwords).

6.3.12 (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 **licensePoolId** 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 `freeSoftwareLicenseKey` to release the current licence reservation.

  The syntax is analogous to the syntax for **DemandLicenseKey.**

6.3.13 Retrieving Error Infos from Service Calls

The String function

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"**.

Similarly the function

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
```

6.4 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:

```
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 KEYs may be associated with identical VALUE parts).

6.4.1 Info Maps

- **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
  ```

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 6.4.4).

- **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)
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_country_code
default_oem_code_page
default_ansi_code_page
default_mac_code_page
```

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 6.4.4). We may now write scripts using a construct like

```
if getValue("language_id_2chars", languageInfo) = "DE"
    ; install German version
else
   if getValue("language_id_2chars", languageInfo) = "EN"
        ; install English version
   endif
endif
```

The function **GetLocaleInfoMap** is meant to replace the older **GetLocaleInfo** is where the delivered values were difficult to interpret:

– **GetLocaleInfo** (DEPRECATED)

  retrieves the (supposedly) most interesting data from the locale data, namely (at this moment)

  - the two-letter version of the system default language name
  - the three-letter version of it (including subtypes of language)
  - the english language name
  - the english country name
  - the language code (hexadezimal value as String)

Usage: If we define and call

```
DefStringList $languageInfo$
set  $languageInfo$ = getLocaleInfo
```

we have a 5 elements String list. In the log file, with the appropriate log level, we get

```
retrieving strings from getLocaleInfo:
    (string   0)DE
    (string   1)DEU
    (string   2)German
    (string   3)Germany
    (string   4)0407
```

We may now construct scripts for conditional statements (cf. section 6.7) like

```
if takeString(0, $languageInfo$) = "DE"
    ; install German version
else
   if takeString(0, $languageInfo$) = "EN"
        ; install English version
   endif
endif
```

6.4.2 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 occurences of **String2.** E.g.,

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

  defines the list

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

- **splitStringOnWhiteSpace (StringVal)**

slices **StringVal** by the "white spots" in it. E. g.

```
set list1 = splitString ("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.

6.4.3 Loading the Lines of a Text File into a String List

- **loadTextFile (filename)**

  reads the file **filename** and generates the String list that contains all lines of the file.

  If the file has unicode format the function

- **loadUnicodeTextFile (filename)**

  should be used. By this call, the strings are converted into the system default 8 bit code.

6.4.4 Simple String Values generated from String Lists

E.g. a spliced string or any transformation of it can be recombined by the function

- **composeString (stringList, linkString)**

  E.g. if **list1** represents the list `'a', 'b', 'c', 'd', 'e'` by

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

  we set the String variable **line** to the value "a | b | c | d | e".

A String value can be retrieved from a list by

- **takeString (index, list1)**

  E. g., if **list1** represents the list of the first five letters of the alphabet by

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

  we get string 'c' (since the index is counted from 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'.

The following function tries to interpret a String list **list1** as list of lines of the form

```
key=value
```

Such,

- **getValue (key, list1)**

  looks for the first line, where the String **key** is followed by the equality sign, and returns the remainder of the line (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 6.4.1 and 6.4.2).

6.4.5 Producing String Lists from wInst Sections

- **retrieveSection (sectionName)**
  gives the lines of the specified section as String list.

- **getOutStreamFromSection (sectionName)**
  invokes the section and – at this moment implemented only for DosBatch, DosInAnIcon (ShellBatch) and ExecPython calls – captures the output to standard out and standard error of the invoked commands writing them into a String list. For example:

  We declare

  ```
  [DosInAnIcon_netuse]
  net use
  ```

  Then the result of

  ```
  getOutStreamFromSection ('DosInAnIcon_netuse')
  ```

  contains among some surrounding stuff the list of all mounted shares of a PC

- **getReturnListFromSection (sectionName)**
  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 (where `XMLPatch_mime` is defined as in section 7.7 in this manual).

Or the list of opsi clients is produced by the reference to the following opsi service call (cf. Section 7.13)

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

where

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

6.4.6 Transforming String Lists

A partial list of a given list is produced by the function:

- **getSubList (startIndex, endIndex, 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'`.

- **`reverse (list)`**

  produces the inverted list, if `list1` is `'a', 'b', 'c', 'd', 'e',` by

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

  we get the list `'e', 'd', 'c', 'b', 'a'`.

6.4.7 Iterating through String Lists

An important application of String lists is based on the device that the script runs through all elements of a list executing some operation on each.

The syntax to define this repetition is:

- **`for %s% in list do statement`**

This expression locally defines a String variable **`%s%`** that takes one by one the values of the **`list`** elements.

`statement` can be any single statement that can exist in a primary section type. In particular (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.

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

```
set line = line + 'a'
```

```
set line = line + 'b'
set line = line + 'c'
set line = line + 'd'
set 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.

For further examples cf. the cook book chapter, e.g. section 8.2.

6.5 Special Commands

- **Killtask <String expression>**

  tries to stop all processes that execute the program named by the String expression.

  E.g.

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

6.6 Commands for User Information and User Interaction

- **Message <String expression>**

  or

- **Message = <sequence of characters>**

  lets **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 "Installing Mozilla Firefox"
  ```

  On the other hand, the command

- **ShowMessageFile <String expression>**

interprets the String expression as text file name, tries to read the text und 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.

The statement

- **ShowBitMap [/<location index>] [<image name>] [<inscription>]**

  places the image denoted by the **image name** (in BMP or PNG format, size 160x160 pixel) at the position denoted by the **location index** and subtitled by the **inscription**.

  <location index> is a <sequence of digits> - in fact at this time there are only positions 1, 2, 3.

  <image name> and **<inscription>** are String expressions.

  E.g. we may call

  ```
  ShowBitmap /3 "%scriptpath%\" + $ProductName$ + ".bmp"  "$ProductName$"
  ```

  for producing a product specific image at window position 3.

  If the name parameter is missing the image at the referred position is cleared.

- **comment <String expression>**

  or

- **comment = <sequence of characters>**

  writes the value of the String expression resp. the sequence of characters into the log file.

  Additional error messages or warnings can be written to the log file by the statements

- **LogError <String expression>**

  or

- **LogError = <sequence of characters>**

  resp.

- **LogWarning <String expression>**

  or

- **LogWarning = <sequence of characters>**

  The following statements are mainly intended for debugging purposes:

- **Pause <String expression>**

  or

- **Pause = <sequence of characters>**

  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.

  On the contrary, the statements

- **Stop <String expression>**

  or

- **Stop = <sequence of characters>**

  are able to end program execution if the user confirms it. The String expression resp. the (possibly empty) sequence of chars explain to the user what is going to be stopped.

- **sleepSeconds <Integer>**

  lets the program execution stop for **<Integer>** seconds

- **markTime**

  Sets a time stamp for the current system time and logs it.

- **diffTime**

  Logs the time passed since the last marked time.

6.7 Conditional Statements (`if` Statements)

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

6.7.1 Example

Recall the example where the script branches dependent on the OS running:

```
DefVar $OS$
Set $OS$ = GetOS
DefVar $NTVersion$

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

  if ( $NTVersion$ = "NT4" ) or ( $NTVersion$ = "Win2k" )
     sub_install_winnt
  else
     if ( $NTVersion$ = "WinXP" )
       sub_install_winXP
     else
       stop "OS version not supported"
     endif
  endif

endif
```

6.7.2 General Syntax

The syntax of the complete **if** statement reads

```
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 **wInst** script).

6.7.3 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 **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 $Name1$ <= $Name2$
```

or

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

For additional examples and some special comparison functions cf. section 6.3.12.

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 (<String expression>)**
  returns *true* if the denoted file or directory exists otherwise *false*.

- **LineExistsIn (line, filename)**
  returns *true* if the text file denoted by **filename** 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 (stringval, filename)**
  returns *true* if there is line that begins with **stringval** 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 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 8.6.

- **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 8.6.

- **HasMinimumSpace (drivename, capacity**)
  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 of use:

```
if not (HasMinimumSpace ("%SYSTEMDRIVE%", "500 MB"))
  LogError "Not enough Space on drive %SYSTEMDRIVE%, required 500 MB"
  isFatalError
endif
```

Helpful for the implementation of the delivery of license keys is the function

- **opsiLicenseManagementEnabled**

It may be used to branch a script depending on the source of a licence key:

```
if opsiLicenseManagementEnabled
          set $mykey$ = DemandLicenseKey ("pool_office2007")
else
          set $mykey$ = IniVar("productkey")
```

6.8 Subprogram Calls

Statements in primary sections which refer to instructions declared elsewhere are called sub program (or procedure) calls.

E.g., the statement

```
sub_install_winXP
```

"calls" the section titled [sub_install_winXP] which is placed somewhere else in the script, as in the example

```
[sub_install_winXP]
Files_copy_XP
WinBatch_SetupXP
```

As long as a sub section, being yet a primary section, is called the chain of reference may continue. In the example program execution jumps first to section `[Files_Kopieren_XP]`, then to `[WinBatch_SetupXP]`.

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

(1) 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).

(2) We may put the referred instructions into *another file* which serves as an *external section*.

(3) *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:

6.8.1 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 ore ore 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 a again a primary section) or **Files** and **WinBatch** (calls of special secondary sections). The complete overview of the existing sub program types is given in chapter 6.

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

  Case (1): The subp rogram is a sequence of lines situated in the executed **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.

  Case (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. **wInst** tries to open the file as a text file and interpret its line 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.

Case (3): If the expression following a stand alone section type specifier is resolvable as a String list expression then the string components 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 7.

6.9 Controlling Reboot

The statement **`ExitWindows`** offers to apply the whole diversity of the underlying system command in a **`wInst`** script.

On principle, **`ExitWindows`** triggers a reboot (resp. an automatic log out or shutdown) after the end of script execution. In the interactive mode the user is asked if she or he agrees with rebooting (at once). If **`wInst`** works in pcprofil mode then the specific ExitWindows request is written to the registry. In an opsi environment, with installed preloginloader, the **`wInst`** process is a subprocess of the execution of **`pcptch.exe`**. When **`wInst`** execution is finished, **`pcptch.exe`** reads the registry entry and calls the system function `exitwindows`. This call does not succeed in Windows XP, therefore the opsi

service process checks the registry again, and enforces the call to `exitwindows`. In batch mode, **wInst** calls the system `exitwindows` command itself.

There are variants of the **ExitWindows** command which trigger a reboot, a logout or a shutdown.

There are two types of a reboot request plus a deprecated one. We list them in the order of increasing urgency of the request:

- **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 **wInst** has finished the currently treated script.

- **ExitWindows /ImmediateReboot**
  breaks the normal execution of a script anywhere inside it. When this command is called **wInst** runs as directly as possible to its end entailing the system `exitwindows` call. In the context of an installed preloginloader it is guaranteed that after rebooting **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.

Logging out instead of rebooting is started – analogously to an "**ImmediateReboot**" – by the command

- **ExitWindows /ImmediateLogout**
  The normal execution of a script breaks at the point of the call, entailing a system log out call.

  This behaviour is needed if an automated user log in for some other user shall take place (cf. cookbook, section 8.3).

Finally, we may demand a shut down at the end of all script executions. For this purpose there is the **/ShutdownWanted** parameter:

- **`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
```

6.10 Keeping Track of Failed Installations

If a product installation fails since errors occur, or if some circumstances prevent the installation script from being successfully executed the script execution should not, as usually in an opsi environment, lead to the product state `installed` but the product state `failed`.

To indicate in a **`wInst`** script that regarding he circumstances the current installation is not successful there is the statement

- **`isFatalError`**

If this statement is called **`wInst`** stops the normal execution of the script and sets the product state to `failed`.

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 occured 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 device by the following script example:

```
markErrorNumber
; Erors occuring 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
```

7 Secondary 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 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.

In detail:

7.1 `Files` Sections

A **`Files`** section mainly offers functions which correspond to copy commands of the underlying operating system. The surplus value when using the **`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.

7.1.1 Example

A simple **`Files`** section could read:

```
[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.

7.1.2 Call Parameters

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%**.

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 sourcepath

   It be equivalent as

   copy sourcepath "**%UserProfileDir%\"**

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 sourcepath targetdir

   is interpreted like:

   copy sourcepath "%UserProfileDir%\targetdir"

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

7.1.3 Commands

In a **`Files`** section the following commands are defined:

- **`Copy`**
- **`Delete`**
- **`SourcePath`**
- **`CheckTargetPath`**
- **`zip`**

`Copy` and **`Delete`** roughly correspond the 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.

`zip` is used to create an archive.

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. 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 recurse into subdirectories.
  - **`e`**
    If there are empty subdirectories in the source path they will be created in the source directory as well.
  - **`v`**
    With 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.

It is checked if a newer version exists in the target directory as well as in the windows and the window system directory.

- **V**

  With version checking, but only with regard to a file the target directory.

- **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**

  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**

  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**

  Existing files are not overwritten.

- **c**

  If a system file is in use, then it can be overwritten only after a reboot. The **wInst** default behaviour is therefore that a file in use will be marked for overwriting after the next reboot, AND the **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**

  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 attributs).

- **Delete [-sfd[n]] <path>**
- **Delete [-sfd[n]] <source(mask)>**

  deletes files and directories. Possible options are (with arbitrary ordering)

  - **s**
    We recurse into subdirectories. Everything that matches the path name or the source mask is deleted.
  - **f**
    forces to delete read only files
  - **d [n]**
    Only files of age **n** days or older are deleted. **n** defaults to 1.

- **SourcePath = <source directory>**
  Sets **<source directory>** as default directory for the following **Copy** and (!) **Delete** commands.

- **CheckTargetPath = <Zieldirectory>**
  Sets **<Zieldirectory>** as default directory for **Copy** command . If the specified path does not exist it will be created.

- **zip [-s] <archive directory> <source mask>**
  The command produces a zip archive file for every file that corresponds to the source mask and puts it in the archive directory. Option `-s` lets recurse into the source subdirectories. (This command was used to produce a special sort of archives when server space was scarce.)

7.2 Patches-Sektionen

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]**.

(Since a patched .ini file is similarly built from sections like the **wInst** script we have to be careful to avoid a denotational mess.)

7.2.1 Example

In times when not everything was written to the registry a file named win.ini played a central role. It can be edited via a **Patches** call: In a primary section, we write

```
Patches_WIN.INI "%SYSTEMROOT%\WIN.INI"
```

and the called section may be defined e.g. for Acrobat Writer:

```
[Patches_WIN.INI]
set [Devices] Acrobat Distiller=winspool,Ne00:
set [Devices] Acrobat PDFWriter=winspool,LPT1:
set [PrinterPorts] Acrobat Distiller=winspool,Ne00:,15,45
set [PrinterPorts] Acrobat PDFWriter=winspool,LPT1:,15,45
set [Windows] Device=Acrobat PDFWriter,winspool,LPT1:
```

7.2.2 Call Parameter

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

7.2.3 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 [<Sektionsname>]**
  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.

7.3 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**

E.g. by

```
[PatchHosts_MyHostsPatch]
setAddr ServerNo1 111.111.111.111
setAlias ServerNo1 myServer
```

we decide that the name `ServerNo1` is resolved as 111.111.111.111, and that any call to the alias `myServer` is directed to the same address.

In detail:

- **setaddr <hostname> <IPaddress>**
  sets the IP address for host **<hostname>** to **<IPaddress>**. If there is no entry for host name as yet it will be created.
- **setname <IPaddress> <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 **<IPadress>**.
- **delalias <hostname> <alias>**
  removes the alias name **<alias>** for the host named **<hostname>** .
- **delalias <IPaddress> <alias>**
  removes the alias name **<alias>** for the host with IP address **<IPadress>**.
- **delhost <hostname>**
  removes the complete entry for the host with name **<hostname>**.

- **delhost <ipadresse>**
  removes the complete entry for the host with IP address **<IPadress>**.

- **setComment <ident> <comment>**
  writes **<comment>** after the comment sign for the host with host name, IP address or alias name **<ident>**.

7.4 IdapiConfig Sections

A **IdapiConfig** section writes parameters in idapi*.cfg files which are used by the Borland Database Engine.

This section type is only available for windows.

The name of the file which is to be treated is given as call parameter, e.g.

```
IdapiConfig_resymesa "c:\idapi\idapi.cfg"
```

An example for a section may be:

```
[IdapiConfig_resymesa]
alias:resabw
driver:dbase
;parametername=parameterwert
TYPE=Standard
PATH=C:\ReSyMeSa\Daten
DEFAULT DRIVER=dbase
setalias
```

Generally we have:

- **alias:<alias name>**
  defines an alias name,

- **driver:<driver name>**
  specifies the driver name.

- **setalias**
  finally writes the data to the configuration file.

Depending on the specific driver there can be any number of settings of form

<parameter name>=<parameter value>

7.5 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 6.7.3).

7.5.1 Example

E.g., for a Mozilla preference file we may set the start page of the browser by a call to the following **PatchTextFile** section:

```
[PatchTextFile_NetscapePref]
GoToTop
FindLine_StartingWith 'user_pref("browser.startup.homepage"'
DeleteTheLine
AddLine 'user_pref("browser.startup.homepage", "http://myhomepage.org");'
```

We can get the same effect more easily since especially for patching the mozilla preference files there is a special command. Using it the example reduces to

```
[PatchTextFile_NetscapePref]
Set_Netscape_User_Pref ("browser.startup.homepage", "http://myhomepage.org")
```

7.5.2 Call Parameter

The text file which is to be treated is given as parameter of the **PatchTextFile** call, e.g.

```
PatchTextFile_prefsjs $mailhome$ + "prefs.js"
```

7.5.3 Commands

We have got two commands especially for patching Mozilla preferences files:

- **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 kept.

- **AddStringListElement_To_Netscape_User_Pref ("<preference variable>", "<add values list>")**
  appends one or more elements to a list entry in the given preference file. It

is checked if a *single* value that shall be added is already contained in the list (then it will not be added).
The command may be used to supplement elements in the list of no proxy entries in prefs.js.

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 *actual 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.

If searching shall certainly start at the top line we have to move the line pointer beforehand. This is done by the command

- **GoToTop**

(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]**

Advancing to the bottom line is done by

- **GoToBottom**

By the following command 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):

- **DeleteTheLine**

There is also a command for deleting all lines which begin with a certain String:

- **DeleteAllLines_StartingWith <search string>**

The lines of the file may be augmented by the following commands:

- **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.

We connect to the file system by some other commands:

- **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.

7.6 LinkFolder Sections

7.6.1 Windows

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 an 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** has the empty string as default.
- If no **icon_file** is specified the program file is selected.
- The default **icon_index** is 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.**

By

- **delete_element <link name>**

we remove a link from the open folder.

A complete folder is removed from the base virtual folder by

- **delete_subfolder <folder path>**

7.6.2 Linux

There are some minor differences to the windows version:

Possible virtual folders are:

```
desktop, startmenu, startup, desktopdirectory, common_startmenu,
common_startup, common_desktopdirectory
```

set_link has the following parameters:

```
name:           // name of link
target:         // path and name of program
parameters:     // call parameters of the program
working_dir:    // working directory of the program
```

```
icon_file:       // path and name of icon file
filename         // name of the desktop file (with ext)
type             // link type (explanation cf. below)
categories       // (opt.) ; separated list of categories
genericName      // (opt.) description (name=mozilla->generic=browser)
```

There is no parameter icon_index.

The parameter `type` is required and shall have one of the following values: `Application, Link, FSDevice, MimeType,`

`categories` may be empty or may contain a semicolon separated list of categories from the following table:

Category	Description
Development	An application for development
Building	A tool to build applications
Debugger	A tool to debug applications
IDE	IDE application
GUIDesigner	A GUI designer application
Profiling	A profiling tool
RevisionControl	Applications like cvs or subversion
Translation	A translation tool
Office	An office type application
Calendar	Calendar application
ContactManagement	E.g. an address book
Database	Application to manage a database
Dictionary	A dictionary
Chart	Chart application
Email	Email application
Finance	Application to manage your finance
FlowChart	A flowchart application
PDA	Tool to manage your PDA
ProjectManagement	Project management application
Presentation	Presentation software
Spreadsheet	A spreadsheet
WordProcessor	A word processor
Graphics	Graphical application
2DGraphics	2D based graphical application
VectorGraphics	Vector based graphical application
RasterGraphics	Raster based graphical application

Category	Description
3DGraphics	3D based graphical application
Scanning	Tool to scan a file/text
OCR	Optical character recognition application
Photography	Camera tools, etc.
Viewer	Tool to view e.g. a graphic or pdf file
Settings	Settings applications
DesktopSettings	Configuration tool for the GUI
HardwareSettings	A tool to manage hardware components, like sound cards, video cards or printers
PackageManager	A package manager application
Network	Network application such as a web browser
Dialup	A dial-up program
InstantMessaging	An instant messaging client
IRCClient	An IRC client
FileTransfer	Tools like FTP or P2P programs
HamRadio	HAM radio software
News	A news reader or a news ticker
P2P	A P2P program
RemoteAccess	A tool to remotely manage your PC
Telephony	Telephony via PC
WebBrowser	A web browser
WebDevelopment	A tool for web developers
AudioVideo	A multimedia (audio/video) application
Audio	An audio application
Midi	An app related to MIDI
Mixer	Just a mixer
Sequencer	A sequencer
Tuner	A tuner
Video	A video application
TV	A TV application
AudioVideoEditing	Application to edit audio/video files
Player	Application to play audio/video files
Recorder	Application to record audio/video files

Category	Description
DiscBurning	Application to burn a disc
Game	A game
ActionGame	An action game
AdventureGame	Adventure style game
ArcadeGame	Arcade style game
BoardGame	A board game
BlocksGame	Falling blocks game
CardGame	A card game
KidsGame	A game for kids
LogicGame	Logic games like puzzles, etc
RolePlaying	A role playing game
Simulation	A simulation game
SportsGame	A sports game
StrategyGame	A strategy game
Education	Educational software
Art	Software to teach arts
Construction	
Music	Musical software
Languages	Software to learn foreign languages
Science	Scientific software
Astronomy	Astronomy software
Biology	Biology software
Chemistry	Chemistry software
Geology	Geology software
Math	Math software
MedicalSoftware	Medical software
Physics	Physics software
Teaching	An education program for teachers
Amusement	A simple amusement
Applet	An applet that will run inside a panel or another such application, likely desktop specific
Archiving	A tool to archive/backup data
Electronics	Electronics software, e.g. a circuit designer
Emulator	Emulator of another platform, such as a DOS emulator
Engineering	Engineering software, e.g. CAD programs
FileManager	A file manager

Category	Description
Shell	A shell (an actual specific shell such as bash or tcsh, not a TerminalEmulator)
Screensaver	A screen saver (launching this desktop entry should activate the screen saver)
TerminalEmulator	A terminal emulator application
TrayIcon	An application that is primarily an icon for the "system tray" or "notification area" (apps that open a normal window and just happen to have a tray icon as well should not list this category)
System	System application, "System Tools" such as say a log viewer or network monitor
Filesystem	A file system tool
Monitor	Monitor application/applet that monitors some resource or activity
Security	A security tool
Utility	Small utility application, "Accessories"
Accessibility	Accessibility
Calculator	A calculator
Clock	A clock application/applet
TextEditor	A text editor
KDE	Application based on KDE libraries
GNOME	Application based on GNOME libraries
GTK	Application based on GTK+ libraries
Qt	Application based on Qt libraries
Motif	Application based on Motif libraries
Java	Application based on Java GUI libraries, such as AWT or Swing
ConsoleOnly	Application that only works inside a terminal (text-based or command line application)

7.7 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/).

wInst offers **XMLPatch** sections for editing XML documents. When calling an **XMLPatch** section the document path name is given as parameter, e.g.

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

With the actions defined for this section type **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

To clarify the working of the section commands some concepts shall be sketched:

7.7.1 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 **wInst**, selection based on criteria (1) to (3) and (7) is implemented:

7.7.2 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 **wInst** command is

- **OpenNodeSet**

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

(i) Explicit Syntax

The more explicit syntax may be seen in the following example (for a more complex example cf. the cook book, section 8.4):

```
openNodeSet

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

(ii) Short Syntax

The same node set is given by the line

```
openNodeSet 'define /handler extension="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.

(iii) 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
```

(iv) 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_nodecount_greater_1 false
- warning_when_nodecount_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.

7.7.3 Patch Actions

Their 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"**

E. g.

SetText "rtf"

transforms the element

<fileExtensions>doc<fileExtensions>

into

<fileExtensions>rtf<fileExtensions>

By

SetText ""

we remote the text node completely.

The variant

AddText "rtf"

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

7.7.4 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 *Aktionen* 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.

7.8 ProgmanGroups Sections

This section type is deprecated.

7.9 WinBatch Sections

In a **WinBatch** section any windows executable can be started. This includes that – as from Windows explorer – a file of any type for which a program is registered can be directly called.

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

```
%systemdrive%\temp\setup.exe
```

There a several parameters of the **WinBatch** call which determine if (or how long) **wInst** shall be wait for the started programs returning

Default is that **wInst** waits for every initiated process to come back. This behaviour corresponds to the call parameter **/WaitOnClose**. On the contrary, if **wInst** shall proceed while the started processes run in their own threads we have to apply the call parameter **/LetThemGo**.

The are more sophisticated options for special circumstances.

If we do the call with parameter **/WaitSeconds [number of seconds]** then **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 **wInst** continues processing when the waiting time is finished.

Even more special conditions are given by the options

/WaitForWindowAppearing [window title]

resp.

/WaitForWindowVanish [window title]

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

If we know a process name whose ending we have to await we can use

/WaitForProcessEnding program

This can be combined with a timeout setting:

/WaitForProcessEnding program /TimeOutSeconds seconds

Example:

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

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

7.10 DOSBatch/ShellBatch Sections

7.10.1 Windows

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

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.

The shell process is created with the view set to normal. That has the consequence that a command shell window appears which allows user interaction.

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 **Aktionen** section to get a "Hello World" from the DOS echo command:

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

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

The output of the shell commands can be captured by using the String list function **getOutStreamFromSection()** (cf. section 6.4.4).

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.

7.10.2 Linux

Via **DOSBatch** sections, here better called **ShellBatch** sections do the same job in Linux as in Windows with minor differences:

The temporary batch file is generated in `/tmp` and executed in a xterm environment (xterm -e).

The output of the scripts is written to the log file.

7.11 DOSInAnIcon/ShellInAnIcon Sections

7.11.1 Windows

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.

Instead, the output of the script is written to the log file.

7.11.2 Linux

In Linux, the only difference between a **ShellBatch** and a **ShellInAnIcon** section call is that no xterm window is shown for the second.

7.12 Registry Sections

Of course, this section type is only available for Windows.

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

7.12.1 Example

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
```

7.12.2 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.

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:

- 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 (cf. section 5 in this chapter).

- Similarly, the parameter **`/addReg`** declares that the **`Registry`** section syntax is that of an inf-file (as used e.g. for driver installations (cf. section 6 in this chapter).

These not **`wInst`** specific syntactical variants are not defined in this manual since they usually will be generated programmatically.

7.12.3 Commands

The default syntax of a **`Registry`** section is oriented at the command syntax of other patch operations in **`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 **wInst** syntax of the registry path name the elements of a path are separated by single backslashs.

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 .., `)
  REG_MULTI_SZ (String value arrays, in **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 **<listset_user_Rhino.reg 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

  HKEY_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.)

  In Win2k there is the phenomen observed that the path entry can only beset_user_Rhino.reg read and set by a script if there was set some value before. The following workaround makes things to:

  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:

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

where `RegistryPathPath` looks like

```
[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 <registry key>**
  deletes the registry key recursively including all subkeys and contained variables. The registry key is defined as for OpenKey.

  Example:

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

- **ReconstructFrom <file name>**
  (deprecated)

- **FlushKey**
  ensures that all entries of a key are saved to the file backing the in memory registry (is automatically done when closing a key, therefore in particular when a **Registry** section is left).

7.12.4 Registry Sections to Patch "All NTUser.dat"

A **Registry** section called with parameter **/AllNTUserdats** 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 `HKEY_Current_User` (which is 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 pathes are interpreted relatively:

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
```

7.12.5 Registry Sections in Regedit Format

If a **Registry** section is called with parameter **/regedit** the section is not expected in **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_pcbin\\install"
"configurl"="\\\\bonifax\\opt_pcbin\\pcpatch"
"utilsurl"="\\\\bonifax\\opt_pcbin\\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 **`wInst`** registry section.

But, we cannot really have an internal **`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. **`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 converted. This can be done manually, e.g. by a suitable editor. But we may also feed the original file to **`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 **`wInst`** registry section. That is, a path like [HKEY_CURRENT_USER\Software\ORL] is to replaced by the relative [Software\ORL].

7.12.6 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"
```

7.13 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.

7.13.1 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 STRINGEXPRESSION**
- **/username STRINGEXPRESSION**
- **/password STRINGEXPRESSION**

If these parameters, at least the serviceurl, are given **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 **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.

In the case that there we had a connection to a secondary opsi service we may (re)set the connection to the standard opsi service via the option

- **/preloginservice**

7.13.2 Section Format

An opsiServiceCall is defined by its methodname and a list of parameters.

Both are defined in the section body. It has format

```
"method":METHODNAME-STRING
"params":[
        JSON PARAMETER ENTRIES
        ]
```

JSON PARAMETER ENTRIES is a (possibly empty) list of Strings or more complicated Json items (as required by the specified method).

E.g. we may have a section call

```
opsiservicecall_clientIdsList
```

where the required methodname and the (empty) list of parameters is set by

```
[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:

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

The usage of **GetReturnListFromSection** is documented in the String list function chapter of this manual (section 6.4.5)

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"
          ]
```

7.14 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.

Python as a full grown programming language gives definitely more coding options than any internal **wInst** commands, and is as well far more powerful than a command shell program. Therefore it can be recommended to use python for complicated tasks. Especially if data objects shall be communicated to the opsi service a python script is the natural approach since the opsi service is written itself in python, and there has not to any translation of data coding.

7.14.1 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"
```

where the section shall be defined by:

```
[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.

7.14.2 Interweaving a Python Script with the wInst Script

An **execPython** section is actually integrated with the surrounding **wInst** script by four kinds of shared data:

- A parameter list is transferred to the python script.
- Everything which is printed by the python script is written into the **wInst** log.
- The **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 **wInst** script.

An example for the first two ways of interweaving the python script with the **wInst** script is already given above. We extend it to retrieve the values of some **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 Aktionen 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 = GetOutStreamFromSection('execpython_hello -a "opt a“')
```

7.15 ExecWith Sections

ExecWith sections are more general than **ExecPython** 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.

7.15.1 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 **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 **getOutStreamFromSection.**

The first one declares that the backslash in **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 **wInst** is continuing to interpret its script.

7.15.2 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 **wInst** regards such lines as comments and eliminates them before handle).

8 Cook Book

This chapter contains a growing collection of examples showing real wold problems that can be mastered by simple or sophisticated pieces **wInst** scripting.

8.1 Delete a File in all Subdirectories

Since **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
```

where we have got

```
[files_delete_Alt]
delete "%UserProfileDir%\alt.txt"
```

Neverthelesse we document a workaround which could be used in older **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:

```
; here we are in Aktionen section:

; variable for file name
DefVar $deleteFile$ = "alt.txt"

; String list declarations
DefStringList list0
DefStringList list1

; capture the lines produced by the dos dir command
Set list0 = getOutStreamFromSection ('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$\$deleteFile$"
```

8.2 Check if a Specific Service is Running

If we want to check if a specific service (exemplified with "preloginloader") 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.

```
[Aktionen]
DefStringList list0
DefStringList list1
DefStringList result
Set list0=getOutStreamFromSection('DosBatch_netcall')
Set list1=getSublist(2:-3, list0)

DefVar $myservice$
DefVar $compareS$
DefVar $splitS$
DefVar $found$
Set $found$ ="false"
set $myservice$ = "preloginloader"


comment "============================"
comment "search the list"
; for developping loglevel = 3
; loglevel=3
; in normal use we dont want to log the looping
loglevel = -1
for %s% in list1 do sub_find_myservice
loglevel=2
comment "============================"

if $found$ = "false"
   set result = getOutStreamFromSection ("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


[dosinanicon_start_myservice]
net start "$myservice$"


[dosbatch_netcall]
@echo off
net start
```

8.3 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:

```
[Aktionen]
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\productname`).
- 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\productname` 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 **`Aktionen`** 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:

```
[Aktionen]
Registry_del_autologin
; ....

[Registry_del_autologin]
openkey [HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon]
set "DefaultUserName"=""
set "DefaultPassword"=""
```

The **`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 `$localFilesPath$`); among them the installation script that is to be executed in the local user context;
- creates a RunOnce entry in the registry that calls **`wInst`** with the local script as argument;
- reboots in order to make the registry change work;
- when **`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 **`wInst`** call.
    - The **`wInst`** script should now regularly proceed. But at its end, there must be a `ExitWindows /ImmediateReboot` command. Otherwise the desktop would of the administratrive 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 **`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.

To observe:

- 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 **`IniVar`**) . 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.

(Maybe a newer version of the following example can be found under http://www.opsi.org/opsi_wiki/TemplateForInstallationsAsTemporaryLocalAdmin)

```
; Copyright (c) uib gmbh (www.uib.de)
; This sourcecode is owned by uib
; and published under the Terms of the General Public License.

[Initial]
LogLevel=2
ExitOnError=false
ScriptErrorMessages=on
TraceMode=off


[Aktionen]
DefVar $ProductName$
Set $ProductName$ = "softprod"
DefVar $ProductSizeMB$
Set $ProductSizeMB$ = "20"
DefVar $LocalSetupScript$
Set $LocalSetupScript$ = "local_"+$ProductName$+".ins /batch"
DefVar $LockKeyboard$
; set $LockKeyboard$ to "true" to prevent user hacks while admin is logged in
Set $LockKeyboard$="true"
; Set PasswdLogLevel to -2 to prevent passwords to logged (not working yet)
DefVar $PasswdLogLevel$
Set $PasswdLogLevel$="-2"
DefVar $OpsiAdminPass$
DefStringlist $outlist$

; some variables for the sub sections
DefVar $SYSTEMROOT$
DefVar $SYSTEMDRIVE$
DefVar $ScriptPath$
DefVar $ProgramFilesDir$
DefVar $HOST$
DefVar $AppDataDir$
Set $SYSTEMDRIVE$ = "%SYSTEMDRIVE%"
Set $SYSTEMROOT$ = "%SYSTEMROOT%"
set $ScriptPath$="%ScriptPath%"
set $ProgramFilesDir$="%ProgramFilesDir%"
set $Host$="%Host%"
set $AppDataDir$="%AppDataDir%"
; temp is always useful
DefVar $TEMP$
Set $TEMP$= EnvVar("TEMP")
DefVar $Tmp$
set $Tmp$ = EnvVar("TMP")
;Variables for version of the operating system (OS)-Test
DefVar $OS$
DefVar $MinorOS$
set $OS$ = GetOS
set $MinorOS$ = GetNTVersion

DefVar $RebootFlag$
```

```
DefVar $WinstRegKey$
DefVar $RebootRegVar$
DefVar $AutoName$
DefVar $AutoPass$
DefVar $AutoDom$
DefVar $AutoLogon$
DefVar $AutoBackupKey$
DefVar $LocalFilesPath$
DefVar $LocalWinst$

Set $WinstRegKey$ = "HKLM\SOFTWARE\opsi.org\winst"
Set $RebootFlag$ = GetRegistryStringValue("["+$WinstRegKey$+"]
"+"RebootFlag")
Set $AutoBackupKey$ = $WinstRegKey$+"\AutoLogonBackup"
Set $LocalFilesPath$ = "C:\opsi_local_inst"
Set $LocalWinst$ = "c:\opsi\utils\winst32.exe"

if ($OS$ = "Windows_NT")

 if not (($RebootFlag$ = "1") or ($RebootFlag$ = "2"))
 ;========================
 ; statements before reboot


  if not(HasMinimumSpace ("%SYSTEMDRIVE%", ""+$ProductSizeMB$+" MB"))
   LogError "Not enough space left on  C: . "+$ProductSizeMB$+" MB on C:
required for "+$ProductName$+"."
  else

   ; show product picture
   ShowBitmap /3 "%scriptpath%\localsetup\"+$ProductName$+".bmp"
"$ProductName$"

   Message "Preparing "+$ProductName$+" install ..."
   sub_Prepare_AutoLogon

   ; we need to reboot now to be sure that the autologon work

   ; Reboot initialisieren ...
   Set $RebootFlag$ = "1"
   Registry_SaveRebootFlag
   ExitWindows /ImmediateReboot

  endif ; enough space
 endif ; Rebootflag = not (1 or 2)
 if ($RebootFlag$ = "1")
  ;========================
  ; Statements after Reboot
  ; Set new Rebootflag
  Set $RebootFlag$ = "2"
  Registry_SaveRebootFlag
  ; the work statements

  Message "Preparing "+$ProductName$+" install ..."
  Registry_enable_keyboard
  ExitWindows /ImmediateLogout
  ; now let the autologon work
  ; it will stop with a reboot
```

```
 endif ; Rebootflag = 1
 if ($RebootFlag$ = "2")
  ;========================
  ; statements after second reboot
  Set $RebootFlag$ = "0"
  Registry_SaveRebootFlag
  ; This part must be here even if nothing is done
  ; possibly we do some cleanup
  Message "Cleanup "+$ProductName$+" install ..."
  sub_Restore_AutoLogon
  ; This is the clean end of the installation
 endif ; Rebootflag = 2
else
 LogError "We need Windows 2000/XP for installing with temporary local user"
endif

[sub_Prepare_AutoLogon]
; copy the setup script and files
Files_copy_Setup_files_local
; read actual Autologon values for backup
set $AutoName$ = GetRegistryStringValue ("[HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon] DefaultUserName")
; if AutoLogonName is our setup admin user, something bad happend
; then let us cleanup
if ($AutoName$="opsiSetupAdmin")
 set $AutoName$=""
 set $AutoPass$=""
 set $AutoDom$=""
 set $AutoLogon$="0"
else
 set $AutoPass$ = GetRegistryStringValue ("[HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon] DefaultPassword")
 set $AutoDom$ = GetRegistryStringValue ("[HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon] DefaultDomainName")
 set $AutoLogon$ = GetRegistryStringValue ("[HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Winlogon] AutoAdminLogon")
endif
; backup AutoLogon values
Registry_save_autologon
; prepare the admin AutoLogon
;LogLevel="$PasswdLogLevel$"
LogLevel=-2
set $OpsiAdminPass$= RandomStr
Registry_autologon
; create our setup admin user
DosInAnIcon_makeadmin
LogLevel=2
; remove c:\tmp\winst.bat with password
Files_remove_winst_bat
; store our setup script as run once
Registry_runOnce
; disable keyboard and mouse while the autologin admin works
if ($LockKeyboard$="true")
 Registry_disable_keyboard
endif

[sub_Restore_AutoLogon]
; 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")
; restore the values
;LogLevel="$PasswdLogLevel$"
LogLevel=-2
Registry_restore_autologon
LogLevel=2
; delete our setup admin user
DosInAnIcon_deleteadmin
; cleanup setup script, files and profiledir
Files_delete_Setup_files_local
; 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_makeadmin]
NET USER opsiSetupAdmin $OpsiAdminPass$ /ADD
```

```
NET LOCALGROUP Administratoren /ADD opsiSetupAdmin

[Registry_autologon]
openkey [HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon]
set "DefaultUserName"="opsiSetupAdmin"
set "DefaultPassword"="$OpsiAdminPass$"
set "DefaultDomainName"="%pcname%"
set "AutoAdminLogon"="1"

[Registry_runonce]
openkey [HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce]
set "opsi_autologon_setup"="$LocalWinst$ $LocalFilesPath$\$LocalSetupScript$"

[Registry_disable_keyboard]
openkey [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Kbdclass]
; disable
set "Start"=REG_DWORD:0x4
;enable
;set "Start"=REG_DWORD:0x1
openkey [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Mouclass]
; disable
set "Start"=REG_DWORD:0x4
;enable
;set "Start"=REG_DWORD:0x1

[Registry_enable_keyboard]
openkey [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Kbdclass]
; disable
;set "Start"=REG_DWORD:0x4
;enable
set "Start"=REG_DWORD:0x1
openkey [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Mouclass]
; disable
;set "Start"=REG_DWORD:0x4
;enable
set "Start"=REG_DWORD:0x1

[Files_remove_winst_bat]
delete -f c:\tmp\_winst.bat
```

8.4 XML File Patching: Setting Template Path for OpenOffice.org 2

Setting the template path can be done by the following script extracts

```
[Aktionen]
; ....

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="$oooTemplateDirectory$"
```

8.5 Retrieving Values From a XML File

As treated in chapter 7.7 , **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 read:

```
<?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\Un
install\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\Un
install\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:

```
[Aktionen]
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.

8.6 Inserting a Name Space Definition Into a XML File

The **`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. Special Error Messages

9 No Connection with the opsi Service

What the matter if **wInst** reports "... cannot connect to service"?

The information which is shown additionally may give a hint to the problem:

- **Socket-Error #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.