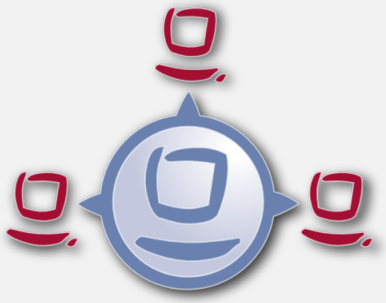

opsi-script: Advanced Scripting

uib gmbh
Bonifaziusplatz 1b
55118 Mainz
Tel.:+49 6131 275610
www.uib.de
info@uib.de

Stand: 2. März 2018

Inhaltsverzeichnis

Kapitel 1

Copyright

Kapitel 2

Übersicht

- opsi-script: Internal flow
  - Fehler Erkennung
  - Fehler in externen Scripten
  - Fehler Behandlung
- opsi-script: Local Functions
  - Local Functions
  - Libraries of Local Functions
  - Missing Features
  - UIB Library of Local Functions
  - Project: Community driven
    opsi.org Library of Local Functions

Kapitel 3

Voraussetzungen

- Laptop mit administrativen Zugriff
- aktueller opsi-winst / opsi-script
- Kentnisse im Umgang mit Stringlisten in opsi-script

Kapitel 4

opsi-script: Internal flow

- opsi-script: Internal flow
  - Fehler Erkennung
  - Fehler in externen Scripten
    - bash
    - cmd.exe
    - powershell
  - Fehler Behandlung

4.1 Errordetection

Methoden:

```
getLastExitCode :string (exitcode) [W/L]
shellCall (<command string>) :string (exitcode) [W/L]
startProcess(<string>) :string (exitcode)  [W/L]

markErrorNumber / errorsOccurredSinceMark [W/L]
LogError
```

4.1.1 Errordetection 1: Exitcode 1

Methoden:

```
getLastExitCode :string (exitcode) [W/L]
```

- Die String-Funktion getLastExitCode gibt den ExitCode des letzten Prozessaufrufs der vorausgehenden WinBatch / DosBatch / ExecWith Sektion aus.
- Der Aufruf anderer opsi-winst Befehle (wie z.B. einer Files Sektion) verändert den gefundenen ExitCode nicht.
- Bei DosBatch und ExecWith Sektionen erhalten wir den Exitcode des Interpreters. Daher muss in der Regel der gewünschte Exitcode in der Sektion explizit übergeben werden.

4.1.2 Errordetection 1: Exitcode 2

(error_exitcode_0.opsiscript)

cmd.exe

```
[ShellInAnIcon_win_exit1]
rem create an errorlevel= 1
echo huhu | findstr ha
exit %ERRORLEVEL%
```

bash

```
[ShellInAnIcon_lin_exit1]
set -x
# create an exit code= 1
echo huhu | grep ha
exit $?
```

4.1.3 Errordetection 1: Exitcode 3

(error_exitcode_1.opsiscript)

cmd.exe

```
[ShellInAnIcon_win_exit1]
rem create an errorlevel= 1
echo huhu | findstr ha
rem create an errorlevel= 0
echo huhu | findstr hu
exit %ERRORLEVEL%
```

bash

```
[ShellInAnIcon_lin_exit1]
set -x
# create an exit code= 1
echo huhu | grep ha
# create an exit code= 0
echo huhu | grep hu
exit $?
```

4.1.4 Errordetection 1: Exitcode 3a

(error_exitcode_2.opsiscript)

cmd.exe

```
[ShellInAnIcon_win_exit1]
set exitcode=0
rem create an errorlevel= 1
echo huhu | findstr ha
if %ERRORLEVEL% NEQ 0 set exitcode=%ERRORLEVEL%
rem create an errorlevel= 0
echo huhu | findstr hu
if %ERRORLEVEL% NEQ 0 set exitcode=%ERRORLEVEL%
exit %exitcode%
```

(error_exitcode_2.opsiscript)

bash

```
[ShellInAnIcon_lin_exit1]
set -x
EXITCODE=0
# create an exit code= 1
echo huhu | grep ha
EC=$?; if [ $EC -ne 0 ]; then EXITCODE=$EC; fi
# create an exit code= 0
echo huhu | grep hu
EC=$?; if [ $EC -ne 0 ]; then EXITCODE=$EC; fi
exit $EXITCODE
```

4.1.5 Errordetection 1: Exitcode 3b

In powershell ist folgende Formulierung hilfreich um (z.B.) den Befehl:
`get-partition | select disknumber, partitionnumber, driveletter, size,type`
auszuführen:

```
Execwith_ps_partitions powershell.exe

[Execwith_ps_partitions]
trap { write-output $_ ; exit 1 }
get-partition | select disknumber, partitionnumber, driveletter, size,type
exit $LASTEXITCODE
```

4.1.6 Errordetection 1: Exitcode 4

Methoden:

```
shellCall (<command string>) :string (exitcode)  [W/L]
```

Führt den Befehl `<command string>` mit der Standard Shell (cmd / bash) aus und liefert den Exitcode als String zurück.

(error_exitcode_3.opsiscript)

Beispiel:

```
set $exitcode$ = shellCall('echo huhu | findstr ha')
```

Ist unter Windows eine Abkürzung für den Ausdruck:

```
DosInAnIcon_netstart winst /sysnative
set $exitcode$ = getLastExitcode

[DosInAnIcon_netstart]
echo huhu | findstr ha
exit %ERRORLEVEL%
```

4.1.7 Errordetection 1: Exitcode 5

(error_exitcode_3.opsiscript)

```
set $exitcode$ = shellCall('echo huhu | grep ha')
```

Ist unter Linux eine Abkürzung für den Ausdruck:

```
shellInAnIcon_ping
set $exitcode$ = getLastExitcode

[shellInAnIcon_netstart]
echo huhu | grep ha || exit $?
```

Nochmal unser powershell Beispiel mit `shellCall`:

```
shellCall('powershell.exe -Command "trap { write-output $_ ; exit 1 } ; get-partition | select
    disknumber, partitionnumber, driveletter, size,type ; exit $LASTEXITCODE"')
```

4.1.8 Errordetection 1: Exitcode 6

Methoden:

```
startProcess(<string>) :string (exitcode)   [W/L]
```

Startet das Programm `<string>` als Prozess und liefert den Exitcode zurück.

Beispiel:

```
set $exitcode$ = startProcess('setup.exe /S')
```

Ist eine Abkürzung für den Ausdruck:

```
Winbatch_setup
set $exitcode$ = getLastExitcode

[Winbatch_setup]
setup.exe /S
```

4.1.9 Errordetection 2: markErrorNumber

Methoden:

```
markErrorNumber : noresult [W/L]
errorsOccurredSinceMark <relation> <integer> : boolean [W/L]
LogError <error - string> [W/L]
```

Beispiel: (error_markerror_0.opsiscript)

```
markErrorNumber
logError "test error"
if errorsOccurredSinceMark > 0
    comment "error occured"
else
    comment "no error occured"
endif
```

4.1.10 Fehlerauswertung 1

Methoden:

```
isFatalError [W/L]
isSuspended [W/L]
isSuccess [W/L]
noUpdateScript [W/L]
```

4.1.11 Fehlerauswertung 2: isFatalError 1

```
isFatalError
isFatalError <short message"
```

Nach dem der Befehl aufgerufen wurde, werden keine Anweisungen mehr ausgeführt und als Skriptergebnis wird *failed* zurückgeliefert. Wird dieser Befehl nicht aufgerufen, so ist das Skriptergebnis *success*.

4.1.12 Fehlerauswertung 2: isFatalError 2

Machmal ist es nicht gewünscht nach dem ersten Fehler abzubrechen, aber trotzdem am Ende das Script mit einem *failed* zu beenden.

Das ist insbesondere bei Umfangreichen Installationen / Konfigurationen der Fall.

```
DefStringList $ErrorList$
DefVar $fatal_error$

ShellInAnIcon_config_depotadmin
if not("0" = getLastExitCode)
  LogError "failed config_depotadmin"
  set $fatal_error$ = "true"
  set $errorList$ = addtolist($errorList$, " failed config_depotadmin")
endif

if count($errorList$) > "0"
  logError "Error summary:"
  for %akterror% in $errorList$ do LogError "%akterror%"
endif

if $fatal_error$ = "true"
  isFatalError
endif
```

4.1.13 Fehlerauswertung 3

- `isSuccess` //since 4.11.3.7 [W/L]
  Abort the script as successful.
- `noUpdateScript` //since 4.11.3.7 [W/L]
  Do not run a update script after setup even if there is one.
- `isSuspended` //since 4.11.4.1 [W/L]
  Abort the script without notice to the server. The action request remain unchanged.

Kapitel 5

opsi-script: Local Functions

5.1 Was ist das Problem

- Globale Variablen
- Räumliche Trennung von Aufruf und Deklarartion von Sektionen

5.2 Vorstellung: Local Functions in opsi-script

Seit Version 4.12 kennt opsi-script auch lokale Funktionen.

Ein Beispiel:

```
DefFunc myFunc(val $str1$ : string, $str2$ : string) : string
        set $result$ = $str1$ + $str2$
endfunc
```

5.2.1 Konzept

Ziel dieser Erweiterung ist die Umsetzung folgender Konzepte:

- Funktionen mit Rückgabewert:
  Die Funktionen haben einen Rückgabewert welche vom Typ `string` oder `stringlist` ist. Der Aufruf eine solchen Funktion kann überall da erfolgen, wo ein Stringausdruck bzw.eine Stringliste erwartet wird.
- Frei definierbare Aufrufparameter: Einer Funktion können Parameter übergeben werden. Diese Parameter werden bei der Deklaration der Funktion definiert. Die Aufrufparameter können vom Typ `string` oder `stringlist` sein. Die Aufrufparameter können als *CallByValue* oder per *callByReference* übergeben werden. *CallByValue* ist der Default. Das bedeutet: wird keine Aufrufmethode explizit angegeben, so wird *CallByValue* verwendet. Soll *CallByValue* explizit angegen werden, so erfolgt dies über das Schlüsselwort `val`. *CallByValue* bedeutet, das beim Aufruf der Inhalt einer beim Aufruf verwendeten Variable auf die Aufrufvariable kopiert wird.
  CallByReference muß über das Schlüsselwort `ref` explizit angegeben werden. *callByReference* bedeutet, dass beim Aufruf eine Verbindung zwischen der aufrufenden Varibalen und des lokalen Aufrufparameters erstellt wird. Eine Änderung der lokalen Variable des Aufrufparameters, wirkt sich direkt auf die beim Aufruf verwendete Variable aus.
  Die Übergabeparameter stehen innerhalb der Funktion als lokale Variablen zur Verfügung.

- Lokale Variablen:
  Eine Funktion enthält lokale Variablen. Implizit gibt es die Aufrufparameter als lokale Variablen und die Variable `$result$` welche vom Typ des Rückgabewertes ist. Darüberhinaus können weitere Variablen innerhalb der Funktion definiert werden.
  All dies Variablen sind lokal, d.h. sie sind nur innerhalb dieser Funktion sichtbar. Eine lokale Variable mit dem selben Namen einer globalen Variable verdeckt innerhalb der Funktion die entsprechende globale Variable.
- Geschachtelte Funktionen:
  Eine lokale Funktion kann wiederum eine oder mehrerr Definitionen von lokalen Funktionen enthalten. Diese Funktionen sind nur innerhalb der Funktion sichtbar in der sie definiert sind.
- Rekursive Aufrufe:
  Eine Funktion kann sich selbst rekursiv aufrufen.
- Primäre und sekundäre Sektionen innerhalb von Funktionen:
  Der Funktionskörper kann eigene Sektionen enthalten. Dies können sowohl primare (sub) Sektionen sein, als auch sekundäre (Files, Registry, ...) Sektionen. Diese sind lokal zu dieser Funktion also nur innerhalb der Sektion sichtbar.

5.2.2 Syntax

Definition

```
DefFunc <func name>([calltype parameter type][,[calltype parameter type]]) : type
<function body>
endfunc
```

Dabei ist:

- `DefFunc` das Schlüsselwort zur Beginn der Definition einer lokalen Funktion.
- *<func name>* der frei gewählte Name der Funktion.
- *calltype* ist der Aufruftyp [`val` | `ref`]. Wird kein Aufruftyp angegeben, + so gilt `val` als gesetzt.
- *parameter* ist der freigewählte Name des Aufrufparameters, welcher unter diesem Namen innerhalb der Funktion als lokale Variable zur Verfügung steht.
- *type* ist der Datentyp des Parameters bzw. der Funktion und entweder `string` oder `stringlist`;
- *<function body>*: ist der Körper der Funktion, welcher dem opsi-script syntax genügen muß.
  In diesem Teil gibt es die automatisch erzeugte lokale Variable `$result$`, welche den Datentyp der Funktion hat (String/Stringliste) und dazu dient den Rückgabewert aufzunehmen.
- `endfunc` zeigt als Schlüsselwort das Ende einer Funktionsdefinition an.

5.3 Beispiele: Local Functions in opsi-script

Einfache Funktion welche zwei Strings miteinander verbindet:
(defined_functions0.opsiscript)

```
[actions]
DefVar $mystr$
DefVar $str1$
set $str1$ = 'ha'

DefFunc myFunc(val $str1$ : string, $str2$ : string) : string
        set $result$ = $str1$ + $str2$
```

```
endfunc

set $mystr$ = myFunc("he","ho")
set $mystr$ = myFunc("he",timeStampAsFloatStr)
set $mystr$ = myFunc("he",$str1$)
```

Erwartete Ergebnisse:

- *heho*
- *he42921.809*
- *heha*

Funktion vom Type `stringlist` welcher ein `string` und eine `stringlist` übergeben werden:
(defined_functions1.opsiscript)

```
[actions]
DefVar $mystr$
DefVar $str1$
DefStringlist $list1$
DefStringlist $list2$

set $str1$ = 'ha'

DefFunc myFunc1(val $str1$ : string, $list1$ : stringlist) : stringlist
        set $result$ = createStringlist($str1$ , takeString(2,$list1$))
endfunc

set $list2$ = splitstring("/etc/opsi/huhu","/")
set $list1$ = myFunc1("hi",$list2$)
```

Erwartete Ergebnisse:

- $list1$ = [hi,opsi]

Funktion vom Type `string` welcher ein `string` übergeben wird:
(defined_functions2.opsiscript)

```
[actions]

DefFunc myFunc2($str1$ : string) : string
        if $str1$ > "0"
                set $result$ = "true"
        else
                set $result$ = "false"
        endif
endfunc

if stringtobool(myfunc2("1"))
        comment "true"
else
        comment "false"
endif
```

Erwartete Ergebnisse:

- *true*

Funktion vom Type `string` welcher ein `string` übergeben wird mit lokaler Variable:
(defined_functions3.opsiscript)

```
[actions]
DefVar $mystr$

DefFunc myFunc3($str1$ : string) : string
        DefVar $locstr1$
        set $locstr1$ = '123'
        set $result$ = $locstr1$ + $str1$
endfunc

set $mystr$ = myFunc3("he")
```

Erwartete Ergenisse:

- *123he*

Funktion vom Type `string` welcher ein `string` übergeben wird mit lokaler Variable und geschachtelter Funktion:
(defined_functions4.opsiscript)

```
[actions]
DefVar $mystr$

DefFunc myFunc4($str1$ : string) : string
        DefVar $locstr1$

        DefFunc myFunc5($str1$ : string) : string
                set $result$ = 'inner' + $str1$
        endfunc

        set $locstr1$ = '123'
        set $result$ = $str1$ + myFunc5($locstr1$)
endfunc

set $mystr$ = myFunc4("outer")
```

Erwartete Ergebnisse:

- *outerinner123*

Einfache Funktion vom Type `string` welcher ein `string` by reference übergeben wird mit lokaler Variable:
(defined_functions6.opsiscript)

```
[actions]
DefVar $mystr$
DefVar $str1$
DefVar $str2$

set $str1$ = 'ha'
set $str2$ = 'hi'

DefFunc myFunc6(ref $str1$ : string) : string
        DefVar $locstr1$
        set $locstr1$ = '123'
        set $str1$ = 'setinlocal'
        set $result$ = $locstr1$ + $str1$
```

```
endfunc

set $mystr$ = myFunc6($str2$)
set $mystr$ = $str1$ + $str2$
```

Erwartete Ergebnisse:

- *123setinlocal*
- *hasetinlocal*

Funktion vom Type `stringlist` welcher eine Variable vom Type `stringlist` mit *call by reference* übergeben wird mit lokalen `stringlist` Variable:
(defined_functions7.opsiscript)

```
[actions]
DefVar $mystr$
DefStringlist $list1$
DefStringlist $list2$

set $list2$ = splitstring("/etc/opsi/huhu","/")

DefFunc myFunc7(ref $list1$ : stringlist) : stringlist
        DefStringlist $loclist1$
        set $loclist1$ = splitstring("/a/b/c","/")
        set $list1$ = createStringList('setinlocal')
        set $loclist1$ = addListToList($loclist1$,$list1$)
        set $result$ = $loclist1$
endfunc

set $list1$ = myFunc7($list2$)
comment "$list2$ index 0: " + takestring(0,$list2$)
```

Erwartete Ergenisse:

- $list1$ = [,a,b,c,setinlocal]
- *setinlocal*

Funktion vom Type `stringlist` welcher ein `string` übergeben wird mit lokaler Variable und lokaler sekundärer Sektion:
(defined_functions8.opsiscript)

```
[actions]
DefStringlist $list1$

DefFunc myFunc8($str1$ : string) : stringlist
        DefStringlist $loclist1$
        set $loclist1$ = getoutstreamfromsection("shellInAnIcon_test")
        set $result$ = $loclist1$

        [shellinanicon_test]
        set -x
        $str1$
endfunc

if GetOS = 'Linux'
```

```
        set $list1$ = myFunc8('set -x ; pwd')
else
        set $list1$ = myFunc8('echo %cd%')
endif
```

Erwartete Ergebnisse: Linux: $list1$ = [+ pwd, /home/uib/gitwork/lazarus/opsi-script]

Windows:

```
The value of the variable "$list1$" is now:
(string    0)
(string    1)W:\opsi-script-test\CLIENT_DATA\standalone-tests>echo W:\opsi-script-test\CLIEN
(string    2)W:\opsi-script-test\CLIENT_DATA\standalone-tests
```

Kapitel 6

opsi-script: Libraries of Local Functions

Hier geht es um die Auslagerung von *local functions* in externe Dateien und um die Möglichkeit diese aus unterschiedlichen Scripten heraus aufzurufen.

6.1 Syntax

`importLib` <string expr> ; import library // since 4.12.0.0
<string expr> : <file name>[.<file extension>][`::`<function name>]
Wenn keine *.<file extension>* (Dateierweiterung) übergeben wird, so wird `.opsiscript` als Default verwendet.
Wenn kein *::<function name>* übergeben wird, so werden alle Funktionen der angegebenen Datei importiert.

<file name> ist:

- Ein kompletter Pfad zu einer Datei. [W/L]
- Eine Datei in `%ScriptPath%` [W/L]
- Eine Datei in `%opsiScriptHelperPath%\lib` [W]
  Entspricht: *%ProgramFiles32Dir%\opsi.org\opsiScriptHelper\lib*
- Eine Datei in `%ScriptPath%/../lib` //since 4.11.5.2 [W/L]
- Eine Datei in `%WinstDir%\lib` [W]

Die Prüfung erfolgt in dieser Reihenfolge. Die erste Datei die gefunden wird, wird genommen.

6.2 Beispiele

Einfache Funktion welche zwei Strings miteinander verbindet:

Die Library Datei myfirstlib.opsiscript:

```
DefFunc myFunc(val $str1$ : string, $str2$ : string) : string
        set $result$ = $str1$ + $str2$
endfunc
```

Das Programm:
(defined_functions_lib1.opsiscript)

```
[actions]
DefVar $mystr$
DefVar $str1$

importLib "%scriptpath%\myfirstlib.opsiscript"

set $str1$ = 'ha'


set $mystr$ = myFunc("he","ho")
set $mystr$ = myFunc("he",timeStampAsFloatStr)
set $mystr$ = myFunc("he",$str1$)
```

Erwartete Ergebnisse:

- *heho*
- *he42921.809*
- *heha*

Die Zeile
`importLib "%scriptpath%\myfirstlib.opsiscript"`
könnte auch folgende Form haben:

- `importLib "myfirstlib.opsiscript"`
  denn `%scriptpath%` ist im Suchpfad;

- `importLib "myfirstlib"`
  denn die extension `.opsiscript` ist der Default;

- `importLib "myfirstlib.opsiscript::myFunc"`
  Würde gezielt nur diese Funktion importieren.

Kapitel 7

opsi-script: Local Functions / Libraries: Missing Features

Folgende bekannte Wünsche sind mpmentan noch offen:

- proceduren (void functions)
  Funktionen die keinen Rückgabewert haben
- function()
  Funktionen die keine Parameter haben
- boolean functions
  Funktionen die (true / false) zurückliefern
  Workaround: Stringfunktion welche ("true"/"false") zurückliefert und diese mit stringToBool() wandeln.
- config: debug_lib
  Das Ausschalten des Logging für Libraries funktioniert noch nicht
- syntax Fehlermeldungen mit richtiger Quellenangabe
  Bei Syntax Fehlern in Libraries wird noch nicht die korrekte Zeile in der korrekten Datei angezeigt
- importlib innerhalb von functions
  Lädt momentan die Library als lokale Funktion

Kapitel 8

opsi-script: UIB Library of Local Functions

opsi-script Library im opsi-winst Paket

Seit opsi-script 4.12.0.13 enthält das Unterverzeichnis `lib` Librarydateien welche von Standard opsi Produkten verwendet werden:

- uib_backend.opsiscript
- uib_bootutils.opsiscript
- uib_strlistutils.opsiscript
- uib_exitcode.opsiscript

Diese Library wird direkt von uib gepflegt und dient hauptsächlich für interne Zwecke.

Kapitel 9

Project: Community opsi-script Library

Wir starten hiermit den Aufbau einer öffentlichen opsi-script Bibliothek die zum Austauch innerhalb der opsi Community dienen soll

9.1 Übersicht und Anforderungen

- Struktur: Gliederung in Dateien nach Funktionsgruppen
- Namespace
- Dokumentation der Funktionen
- Zentrale Versionskontrolle, Bereitstellung und Download
- Test der Funktionen

9.2 Struktur: Gliederung in Dateien nach Funktionsgruppen

- Gliederung in Dateien nach Funktionsgruppen
- Vorschläge ?

9.3 Namespace

Namespace:

- Ob und wie sollen die Library-Dateien am Namen erkennbar sein ?
  z.B. opsi als Post- oder Prefix ?
- Ob und wie sollen die Funktionen aus dieser Library am Namen erkennbar sein ?
  z.B. opsi als Post- oder Prefix ?

9.4 Dokumentation der Funktionen

Damit Funktionen einer Library auch verwendet werden, müssen sie nach Funktion und Syntax auffindbar sein.
D.h. sie müssen dokumentiert und in einer zentralen Dokumentation aufgefürt sein. Dazu wird zur Zeit ein Werkzeug entwickelt, welches auf Basis von Kommentaren innerhalb der Funktion eine Dokumentation in den Formaten HTML, PDF, ASCIIDOC erstellen kann.

9.5 Zentrale Versionskontrolle, Bereitstellung und Download

Wir planen die Bereitstellung der Library in unserem öffentlichen Versionskontrollsystem GIT.
Zur Verteilung soll die Library natürlich auch als opsi-Paket Bereitgestellt werden

9.6 Test der Funktionen

Für die Funktionen müssen Tests bereit stehen: * Diese sind als Beispiele der Dokumentation hilfreich * Überprüfung für die Entwicklenden der Funktionen * Überprüfung für die zentrale opsi-script Entwicklung