ITS Environment

The ITS acronym refers to IT Scripting, where the IT is a generic department name.

This environment has been slowly developing over the last 10 years, where the author made a move towards Unix and Windows environments being used for serious production purposes.

It is intended to release some parts of the ITS environment to the Open Source community.

Principles

While the Unix and Windows versions of ITS are not identical, they are built with the following **principles** in mind.

1. The file system that implements ITS should **separate code, data and logfiles.** This allows us to logically separate generally read only files from scratch areas, giving file access privileges as required.
2. The scripting environment should **use relative file system paths**. As the first principle forbids the use of hard coded variables, this principle indicates that there should no hard-coded file system paths in any script. Unix and Windows provide environment variables which which can replace all or part of a file specification.
3. If a function or procedure is called from more than once, **the common function should be included from a library.**
4. All scripts should be based on a **standard template.** A standard template will allow developers to become productive quickly. The template should only contain code that does *not* exist in a common function library.
5. The scripting environment should be **data driven.** Scripts should be developed and tested away from production environments. When customisation is needed in a production environment, this customisation should be achieved by modifying tables of data that are read by scripts. We should never alter a scripts contents in production environments.
6. Scripts should employ **common logging and alerting** methods. As most scripts are not run interactively, locating and interpreting their output should be easy. When failures occur that require attention, standard means should be available to notify support staff.
7. Scripts should employ **in-line error handling** where ever possible and **provide a default error handler** as a catch all.
8. Scripts should contain sufficient comments to **generate their own documentation**.
9. Where possible, scripts should be written in a **standard shell or language**. On Windows, the Windows Scripting Host environment is used, which all code written in VBScript. Experience indicates that Microsoft Power Shell is still many years away from supplanting VBScript. On Unix, the Bourne again shell (bash) was chosen as a preferred shell on various distros of Linux.
10. Script filenames should be **as descriptive as possible**. Where a script serves several purposes, file system links should represent the variations.
11. Scripts should have a **standard means of collecting command line arguments.**
12. Scripts should be capable of writing **verbose debugging messages** as required, with the level of verbosity being adjustable at run time.

ITS on Microsoft Windows

The ITS environment for windows is written for Win2k3 and later on servers, XP and later on desktops.

File system structure

ITS on windows uses the following environment variables:

1. ITS_PROCEDURES – refers to where scripts are deployed to and executed from.
2. ITS_DATA – refers to any input or output data that a script reads or writes.
3. ITS_LOG - is a scratch area where logfiles are written.

To deploy ITS on a windows platform, one should create System Environment variables similar to the following example:

```
ITS_DATA=c:¥its¥its_data
ITS_LOG=c:¥its¥its_log
ITS_PROCEDURES=c:¥its¥its_procedures
```

Note: Windows supports Process Environment variables which could be used to run multiple environments on a given platform

The above structure implements **principle 1.**

Run time environment

Vbscript is made available by the Windows Script Host (WSH) environment, which has been available on Windows for over 10 years.

Earlier versions of WSH lacked the capability to Include external modules. The ITS environment uses the Windows Script File (.WSF) method of executing script, which uses XML tags to control execution and allows inclusion of external modules.

Here is a sample of the WSF XML at the beginning of a standard script:

```
<job id="sms2group.wsf">
<script language="VBScript" src="StaticDataDefinitions5.vbs"/>
<script language="VBScript" src="DynamicDataDefinitions5.vbs"/>
<script language="VBScript" src="CommonFunctionsLibrary5.vbs"/>
<script language="VBScript">
```

1. A Windows Script File (WSF) can contain multiple Jobs, tagged by unique Job ID. In general, ITS contains only one job per file
2. Static Data Definitions. A scripting environment will require various **constants** and other variables that always have the same value. This file contains those definitions.
3. Dynamic Data Definitions. Scripts will require variables that are set at run time, prior to mainline initialisation. An example is the **username** of the process executing the script, which unknown until execution. Further, the **environment variables** that implement **principle 2** are set as dynamic data.
4. Common Functions Library. Implementing **principle 3,** this statement includes a set of functions used by many scripts.
5. The last tag indicates that the entire module is written in Vbscript. (WSH supports tagged sections in various script languages)

Initialisation

As any script begins execution, local constants and variables are declared immediately after the Includes section.

The very first procedure called by any script is a subroutine in commonfunctionslibrary called **initialise().**

The **initialise** routine interprets the script name, builds a date and time stamp and creates a **log file name**, of the form ITS_LOG**scriptname_username_yyyymmddmmhhss.log**

All subsequent script output is written to this logfile by the **message** common subroutine. The **message** subroutine calls **logger** common function. **Logger** creates a **log file** as required and appends all **message** output to this **log file**, along with an execution timestamp.

These routines implement **principle 6** via the first examples of **principle 3**.

Error Handling

VBScript has quite simplistic error handling. One of two error handling **modes** are in effect:

1. **On error goto 0 -** this is the default mode, where any run-time error causes an exception which will cause the script to exit. Upon exit, WSH will print the exception details and a **line number**, giving a trouble-shooter a starting point to debug a problem.
2. **On error resume next -** in this mode, the script processor ignores run time errors. The developer must examine the **err** object and make a programmatic decision on how to handle the error

What this means in practice, is we cannot satisfy all of **principle 7** as Vbscript has no concept of a user-defined default error handler. It is possible that any script could simply "fail" and it is difficult to even log the event that lead to failure.

Strictly speaking, any program statement that *could* fail should be wrapped in a code block such as:

```
on error resume next
' Create HTTP object
Set Http = CreateObject("Microsoft.XmlHttp")
if err.number <> 0 then
message "Failed to create MS XMLHTTP object"
on error goto 0
exit function
end if
```

As you can imagine, this could make any code extremely voluminous and provide a high likelihood that an error may be missed. We must accept this limitations of Vbscript and find a compromise in this behaviour through a general approach.

In the above example, the code snippet will perform an **exit function** when an error condition exists. Throughout the ITS environment, function calls are generally checked as follows:

```
If not FuctionName(p1,p2) then
' do something
...
```

Within any function, the general approach is to do in-line error handling in a granular fashion, and use Boolean function returns similar to:

```
on error resume next
' do something
' then check err object
if err.number <> 0
' exit the function with no return value (false)

' at the end of the function, set a return value
FunctionName = True
end function
```

This approach provides a reasonable compromise with the available error handling methods.

Creating a new script

All new scripts should be created from a standard template, as stated in **principle 4**.

This script is customised by a utility called **newscript.vbs**, which *does not* comply to all the other principles, but simply enforces **principle 4** on all new scripts.

Here is the current template WSF file, which contains comment templates (providing the building blocks for **principle 8**), file includes, initialisation and command line argument handling (as specified in **principle 11)**

```
<job id="script_name_token">
<script language="VBScript" src="StaticDataDefinitions5.vbs"/>
<script language="VBScript" src="DynamicDataDefinitions5.vbs"/>
<script language="VBScript" src="CommonFunctionsLibrary5.vbs"/>
<script language="VBScript">
' ================================================================================
' Script.........: [script_name_token.wsf]
' Author.Email...: [ben.burke@internode.on.net]
' Version........: 1.0
' Date Written...........: date_token
'
' One Line Description: one_line_token
'
'
' For Detailed Description and revision history, go to end of file
' (this saves the interpreter from 'reading' the comments.
Option Explicit
On Error Goto 0
' --------------------------------------------------------------------------------
' Declare Constants
' --------------------------------------------------------------------------------
' --------------------------------------------------------------------------------
' Declare Variables
' --------------------------------------------------------------------------------
' --------------------------------------------------------------------------------
' Do something
' --------------------------------------------------------------------------------
initialise
CommandLineArgs
Main
' --------------------------------------------------------------------------------
```

```
' ------------------------------------------------------------------------
' Subroutine.....:
' Purpose........:
' Arguments......:
' Example........:
' Requirements...:
' Notes..........:
' ------------------------------------------------------------------------
sub Main
end sub
' ------------------------------------------------------------------------
' ------------------------------------------------------------------------
' Subroutine.....:
' Purpose........:
' Arguments......:
' Example........:
' Requirements...:
' Notes..........:
' ------------------------------------------------------------------------
' ------------------------------------------------------------------------
' ------------------------------------------------------------------------
' Subroutine.....: CommandLineArgs
' Purpose........: Turn Command line arguements into variables
' Arguments......: None
' Example........:
' Requirements...: Variables need to be declared in higher scope
' Notes..........: Plan to generalise this and move to function library
' ------------------------------------------------------------------------
Sub CommandLineArgs()
dim NamedArgs
set NamedArgs = wscript.arguments.named
if NamedArgs.exists("debug") then
debuglevel = NamedArgs.item("debug")
message "Debuglevel set to " & debuglevel
end if
end sub
' ------------------------------------------------------------------------
' ------------------------------------------------------------------------
' Function.......: function_name
' Purpose........:
' Arguments......:
' Returns........:
' Example........:
' Requirements...:
```

```
' Notes.....:
' ----------------------------------------------------------------------
' ======================================================================
' End Of Script
' ======================================================================
' Description....: ' script_name_token.vbs
'
' Lengthy Description here.
'
'
' Notes.....:
'
' Customize.....:
' ======================================================================
' Revised By.....:
' Email.....:
' Revision Date..:
' Revision Notes.:
'
' ======================================================================
' ======================================================================
</script>
</job>
```

The above script template contains the string “token” in several places. The **newscript.vbs** utility replaces these tokens with some calculated and prompted equivalences.

As noted above, the new script utility is *not* a standard script. We simply prompt the user for a script name and a one line description. Then, we read the standard template, replacing several named tokens. Then, the newly created script is ready for editting.

Here is the **newscript** utility:

```
'Newscript - read a template and fill in some blanks by asking some questions
option explicit
Dim fso
dim oFile
dim i
const template="templatev5.wsf"
main
sub main
dim tso1
```

```
dim tso2
dim userArray
dim newscriptname
dim oneliner
dim YourAPieceOfString
set FSO = wscript.createobject("scripting.filesystemobject")
if fso.fileexists(template) then
set tso1 = fso.opentextfile(template)
YourAPieceOfString = tso1.readall
tso1.close
set tso1 = nothing
else
wscript.echo "Can't find " & template
end if
newscriptname = inputbox ("New script name (no extenstions)-->", "Create a new script")
if newscriptname = "" then exit sub
newscriptname = newscriptname & ".wsf"
if fso.fileexists(newscriptname) then
wscript.echo "Sorry, that file exists " & newscriptname
exit sub
end if
oneliner = inputbox ("Enter a very brief description-->", "One Line description")
YourAPieceOfString = replace(YourAPieceOfString,"script_name_token", newscriptname)
YourAPieceOfString = replace(YourAPieceOfString,"date_token", now())
YourAPieceOfString = replace(YourAPieceOfString,"one_line_token", oneliner)
wscript.echo "Creating " & newscriptname
set tso2 = fso.createtextfile(newscriptname)
tso2.write YourAPieceOfstring
tso2.close
end sub
```

Debugging Levels

A new script contains one standard command line argument, **debug**. This argument is parsed into a variable, which is examined by the **debugwrite** common subroutine. Calls to the **debugwrite** routine look as follows:

```
debugwrite 2, "Now doing action blah blah"
```

When the **debug** command line argument is set to 2 or higher, **debugwrite** will call the **message** routine, passing the second argument.

This way, the granularity of debugging messages can be controller at run time, as per **principle 12**

Access to data

Many implementations based on the ITS environment make extensive use of the Activex Data Object known as a record set.

The common functions library contains functions to connect to various data sources, which include excel spreadsheets, CONNIX data sources, MS-SQL databases and Active Directory objects.

Scripts that apply **principle 5** can use the MakeRecordSet function along with the specific connection routines to access tabular data. (The standard File System Object can be used to read text files that contain data)

ITS on Unix

The ITS environment was developed on Ubuntu Lucid Lynx (10.4) and Open SuSE 11.2, using the Bourne again shell (bash).

File system structure

ITS on unix uses the following environment variables:

1. ITS_SCRIPTS – refers to where scripts are deployed to and executed from.
2. ITS_DATA – refers to any input or output data that a script reads or writes.
3. ITS_LOG - is a scratch area where logfiles are written.
4. ITS_ENV – containing environment setup scripts

To deploy ITS on a unix platform, one should create a directory hierarchy similar to the following:

```
/its
/its/scripts
/its/logs
/its/data
/its/env ’ optional
```

Within the /its/env directory, a default environment setup script can be created, similar to the follow:

```
#!/bin/bash
if [ "$ITSROOT" = "" ]; then
export ITSROOT="/its"
fi
if [ "$DEBUG" = "true" ]; then
set -xv
fi
export ITSS=$ITSROOT/scripts
export ITSL=$ITSROOT/logs
export ITSD=$ITSROOT/data
export ITSE=$ITSROOT/env
PATH="$PATH:$ITSS"
```

The above structure implements **principle 1.**

Run time environment

Since various Unix implementations and shells have different means of implementing run commands (.rc files for a shell), an environment setup independent of platform is required.

As most of these scripts will execute via cron, the environment setup must be universally suitable to the manner in which cron daemons inherit environment variables.

The following approach was taken. A single environment variable ITSCOMMONFUNCTIONS must exist and point to the function library script.

Within a cron tab, the declaration of ITSCOMMONFUNCTIONS will look as follows:

```
# crontab for root
ITSCOMMONFUNCTIONS=/its/scripts/common_functions_library
```

This method has proved workable for all the Unix platforms tested and will allow different environments to co-exist.

Interactive users might execute a setup script similar to:

```
export ITSCOMMONFUNCTIONS=$ITSS/common_functions_library
```

Principles 2 and 3 are implemented by the above approach.

Initialisation

If the function library (pointed to by ITSCOMMONFUNCTIONS) does not exist as a file, execution terminates.

All scripts begin execution by including the function library with a **source** statement.

When the parent process is cron, stdout is redirected to a **logfile** named $ITSL/**scriptname.log**

All subsequent script output is written to stdout by the **generate_event** script, which has the following aliases:

```
gen_info="generate_event -f $PROGNAME -s i -m"
gen_success="generate_event -f $PROGNAME -s s -m"
gen_warning="generate_event -f $PROGNAME -s w -m"
gen_error="generate_event -f $PROGNAME -s e -m"
gen_fatal="generate_event -f $PROGNAME -s f -m"
```

These aliases implement **principle 6** via the first examples of **principle 3**.

Error Handling

Bourne again shell provides a **trap** statement that can be used to invoke a function upon receipt of various signals (NOHUP, INT) or the ERR signal.

Inline error handling should be dealt with by suppressing the **trap** for ERR as follows:

```
# turn off default handler
trap -ERR
# do something and do inline checking of return status (${?})
# turn back on default handler
trap ″error_exit ERR″ ERR
```

The common functions library provides a default error handler that will use standard **generate_event** calls to communicate error conditions, satisfying all of **principle 7.**

Creating a new script

All new scripts should be created from a standard template, as stated in **principle 4**.

At the time of writing, the **newscript** bash script is incomplete. Instead of a template, here is an example script, **timed_rsync**

```
#!/bin/bash
# ----------------------------------------------------------------
#
# Shell program to implement standardised rsync operations based on parameters.
#
# Copyright 2009, Ben Burke,Sydney Australia,Infrastructure Manager - the Virtualco, ben.burke@internode.on.net.
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License as
# published by the Free Software Foundation; either version 2 of the
# License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# General Public License for more details.
#
# Description:
#
#
#
# Usage:
#
# timed_rsync [ -h | --help ] [-s source_path] [-o output_path] [-d]
#
# Options:
#
# -h, --help Display this help message and exit.
# -s source_path source path - rsync will change dir to this location as source
# -o output_path output_path - destination of the rsync
# -d when d is 'yes', actually run the rsyn. Otherwise do a check pass
# -r adds the --remove-source-files switch to the rsync command
#
# Revision History:
#
# 8-dec-2009 File created by new_script ver. 2.1.0
```

```
# 4-Jan-2009 Updated to use standard common_functions_library
# Removed trap ERR and built case statement around rsync return values
# Rsync output directed to TEMP_FILE1.
# Generate event updated to add a files contents to mail message body
#
# ------------------------------------------------------------------
if [ -f "$ITSCOMMONFUNCTIONS" ];then
source $ITSCOMMONFUNCTIONS
else
echo "ITSCOMMONFUNCTIONS not defined. Exitting"
exit 1
fi
# ------------------------------------------------------------------
# Constants
# ------------------------------------------------------------------
# PROGNAME=$(basename $0)
# PROGNAME is now determined in the common_functions_library
VERSION="1.0.1"
# ------------------------------------------------------------------
# any script specific clean up - will be called by included function clean_up
function local_clean_up
{
$gen_info "Local Clean up called"
cd $ORIGINAL_DIR
}
function usage
{
# ------------------------------------------------------------------
# Function to display usage message (does not exit)
# No arguments
# ------------------------------------------------------------------
echo "Usage: ${PROGNAME} [-h | --help] [-s source_path] [-o output_path] [-d] [-r]"
}
function helptext
{
# ------------------------------------------------------------------
# Function to display help message for program
# No arguments
# ------------------------------------------------------------------
local tab=$(echo -en "¥t¥t")
cat <<- -EOF-
${PROGNAME} ver. ${VERSION}
This is a program to implement standardised rsync operations based on parameters.
$(usage)
```

```
Options:
-h, --help Display this help message and exit.
-s source_path source path - rsync will change dir to this location as source
-o output_path output_path - destination of the rsync
-d when -d is present, actually run the rsyn. Otherwise do a check pass
-r adds the --remove-source-files switch to the rsync command, deleting the files that were copied
from the source
-EOF-
}
# ----------------------------------------------------------------
# Program starts here
# ----------------------------------------------------------------
##### Initialization And Setup #####
initialise
##### Command Line Processing #####
if [ "$1" = "--help" ]; then
helptext
graceful_exit
fi
# empty variables to inherit values from the command line
SOURCE_PATH=""
OUTPUT_PATH=""
# Default action is to do test run only - don't actuall copy anything (that's the -n)
DOIT="-n -i"
ORIGINAL_DIR="$(pwd)"
REMOVESOURCE=""
while getopts ":hs:o:dr" opt; do
case $opt in
s ) $gen_info "source path - rsync will change dir to this location as source - argument = $OPTARG"
SOURCE_PATH="$OPTARG"
;;
o ) $gen_info "output_path - destination of the rsync - argument = $OPTARG"
OUTPUT_PATH="$OPTARG"
;;
d ) $gen_info "d is present, we will actually run the rsync"
DOIT="-i"
;;
r ) $gen_info "r is present, we will delete the source files after they are copied"
REMOVESOURCE="--remove-source-files"
;;
h ) helptext
graceful_exit ;;
* ) usage
clean_up
```

```
exit 1
esac
done
##### Main Logic #####
# SOURCE_PATH is a directory - that's about as much checking as we can do
if [ -d "${SOURCE_PATH}" ]; then
cd $SOURCE_PATH
#working with Mark's advice, all rsync will be executed with . as input spec - get around relative path issues
$gen_info "Successfully changed directory to $SOURCE_PATH"
else
$gen_warning "Source Path $SOURCE_PATH is not a directory"
exit 1
fi
# OUTPUT_PATH is a non-blank - since the output path is often remote, can't check too much
if [ "${OUTPUT_PATH}" != "" ]; then
$gen_info "Output to $OUTPUT_PATH "
else
$gen_warning "$OUTPUT_PATH is not set"
clean_up
exit 1
fi
# if we leave the standard ERR trap engaged, we'll fall out through the default error_exit and
# loose the context of what went wrong.
# Turn it off now and leave it off for the rest of the script
trap - ERR
# Will leave debugging output on for the rest of this script during bed in period
set -xv
time rsync $DOIT --no-checksum --no-perms -S -R -a -u -W $REMOVESOURCE . $OUTPUT_PATH > $TEMP_FILE1 2>&1
STATUS=$?
# dump the output of rsync command here too, so it ends up in the logfile
cat $TEMP_FILE1
case $STATUS in
0) graceful_exit;;
1) STATUSMESSAGE="Syntax or usage error";;
2) STATUSMESSAGE="Protocol incompatibility";;
3) STATUSMESSAGE="Errors selecting input/output files, dirs";;
4) STATUSMESSAGE="Requested action not supported: attempt made to manipulate 64-bit files on wrong platform";;
5) STATUSMESSAGE="Error starting client-server protocol";;
6) STATUSMESSAGE="Daemon unable to append to log-file";;
10) STATUSMESSAGE="Error in socket I/O";;
11) STATUSMESSAGE="Error in file I/O";;
12) STATUSMESSAGE="Error in rsync protocol data stream";;
13) STATUSMESSAGE="Errors with program diagnostics";;
14) STATUSMESSAGE="Error in IPC code";;
```

```
20) STATUSMESSAGE="Received SIGUSR1 or SIGINT";;
21) STATUSMESSAGE="Some error returned by waitpid()";;
22) STATUSMESSAGE="Error allocating core memory buffers";;
23) STATUSMESSAGE="Partial transfer due to error";;
24) STATUSMESSAGE="Partial transfer due to vanished source files";;
25) STATUSMESSAGE="The --max-delete limit stopped deletions";;
30) STATUSMESSAGE="Timeout in data send/receive";;
35) STATUSMESSAGE="Timeout waiting for daemon connection";;
*) STATUSMESSAGE="Unknown rsync return status $STATUS";;
esac
# sample of a call to generate event including file to include in emails
generate_event -f $PROGNAME -s e -m "$STATUSMESSAGE - see logfile or mail message contents" -e $TEMP_FILE1
clean_up
exit 1
```

Debugging Levels

ITS for Unix implements debugging levels by calls to generate_event, which takes 5 severity levels. This way, the granularity of debugging messages can be controller at run time, as per **principle 12**

Access to data

Scripts that apply **principle 5** include **cifs_rsync** which reads it's authentication information from files in ITS_DATA