Scripts.hyper

COLLABORATORS			
	TITLE : Scripts.hyper		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 5, 2023	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Scripts.hyper

1.1 Scripts (Wed Jul 15 08:25:12 1992)

1.2 Scripts : Commands used in this tutorial

assign	Assign to a variable for use in ARexx
attach	Attach a command to a key (make a macro)
awin	Open/close 'Rexx' logical window
cleanup	Free all memory allocated with 'alloc'
front	Bring PowerVisor to the front
go	Jump to memory
hide	Hide output from ARexx script
list	List a list
load	Load a file in memory
pvcall	Call PowerVisor internal function
rematttach	Remove a macro
remvar	Remove a variable
resident	Make a ML-script resident
rx	Start an ARexx script
script	Start a script

```
string      Convert a PowerVisor string pointer to an ARexx string
sync        Synchronize ARexx with PowerVisor
unhide      Unhide output from ARexx script
unresident  Remove a resident ML-script
unsync      Undo synchronization of ARexx with PowerVisor
vars        Show all variables
void        Evaluate expressions
```

1.3 Scripts : Functions used in this tutorial

```
alloc       Allocate memory
eval        Evaluate string
free        Free memory
if          Conditional evaluation
```

1.4 Scripts : Introduction

This tutor file explains everything (or almost everything at least) that there is to know about scripts for PowerVisor. This is already quite a lot. ARexx scripts, PowerVisor scripts and machinelanguage scripts are all explained here.

1.5 Scripts : PowerVisor scripts

PowerVisor scripts are the simplest. A PowerVisor script is simply a bunch of PowerVisor commands put after each other. You can't do special things in scripts like goto's, subroutines, if-then-else structures, ...
There are a few things that must be noted though :

- You can put comments in your script files by putting a ';' in front of the line (you may put spaces in front of the ';')
  Note that you can't put comments after a command, unless you are sure that the command will ignore the extra argument (most commands ignore everything after the last argument they need, but there are exceptions)

```
    ;  this is a comment
```

- The quit command works different in scripts. The 'quit' command stops the script. (PowerVisor will not quit)
- The script command does not work in scripts. You can't recursively execute scripts

You can start a script with the 'script' command.

The recommended place for scripts is the s:pv subdirectory but PowerVisor will first try the current directory. When a script is located in the current directory or in the s:pv directory you need not specify the full pathname.

There is one exception. The PowerVisor-Startup script always resides in the s: directory. This is the script that is executed everytime you start PowerVisor. The standard PowerVisor-startup script provided with this release of PowerVisor defines some aliases (extra commands) and makes some macros (key attachements) for you.

There are some commands and functions present in PowerVisor that can be used from the PowerVisor commandline but are more useful in scripts. Two of these functions are if() and eval() . They are explained more fully in the Expressions chapter.

Of course, commands like print and locate are also more useful in scripts.

For a fully documented script example, look at Source/mkeys.explained. This script installs a memory viewer. Don't be alarmed at the apparent complexity of this script (well, you may alarm yourselves a bit, because it is a complex example :-), once you got the feeling you will find it rather easy to use scripts in general.

If you use variables in your script it is recomended that you put an underscore ('_') in front of your variable. That way you minimize the chances for variable collision with user variables. If you are ready with the variable use remvar to remove it.

1.6 Scripts : ARexx scripts

For more complex scripts you can use ARexx. With ARexx you can interface PowerVisor to all other programs that use ARexx. If you want you could even write an ARexx script to debug programs from within your favorite editor, or edit files from within PowerVisor :-)

The PowerVisor ARexx port is called 'REXX_POWERVISOR'. This port is the default ARexx port if your script is started from PowerVisor. But if you want to make global scripts (scripts that can be started from anywhere, like the Shell or an editor) you must use :

```
ARexx< address rexx_powervisor
```

before you issue any PowerVisor command.

You can of course also use :

```
ARexx< address rexx_powervisor 'some powervisor command'
```

(Note that an ARexx script always starts with a comment /* ... */)

Almost all PowerVisor commands can be used from within an ARexx script. There are some differences compared with the commandline :

- You can't use abbreviations for commands, you must always type the full commandname
- You can't use prefixes like '-' and '~' before a command. ('~' is not useful). If you want to hide output ('-'), you must use the

hide command provided for that purpose
- The quit command does not work
- You get some new commands (assign , hide , unhide , sync , string , async and front) (explained below). These commands also work on the PowerVisor commandline but are not very useful there
- Functions are called the same way as commands. The result is put in the 'result' variable (if you use 'options results' in the ARexx script)
- Note that some commands and functions return a string instead of a number when called from ARexx (See the Command Reference and Function Reference chapters for all commands and functions returning strings)

You can use the rx command to start an ARexx script. This command starts the script asynchronous. This means that while the script is running you can still use PowerVisor commands. If you want to disable this feature you can use the sync command from within the ARexx script. This command synchronizes PowerVisor with the ARexx script. You will not be able to use the PowerVisor commandline. When the ARexx script is ready, it should call async . If the ARexx script forgets the 'async' command, you will not be able to use PowerVisor anymore. You can solve this by sending the command 'async' from the Shell (with the 'rx' shell command) :

```
Shell< rx "address rexx_powervisor async" <enter>
```

Note that the default file extension for PowerVisor ARexx scripts is '.pv'. You do not need to type this extension when you use the PowerVisor 'rx' command.

When you execute a PowerVisor command or function from within ARexx and there is an error (or the command is interrupted) you can examine the returncode (the 'rc' variable). 'rc' will contain 0 if there was no error or the PowerVisor error code if there was an error. You can find all PowerVisor error codes listed with the geterror() function.

Because you can't use the '-' prefix to hide output for a command there is another way to hide output. The hide and unhide commands are provided for this. After 'hide' all output from the ARexx commands is hidden (as if there was a '-' in front of the commandline). 'unhide' restores this situation. Note that you must use 'unhide' otherwise you might confuse yourselves when you use other ARexx scripts.

Normally the output for the ARexx scripts goes to the current logical window ('Main' or 'Extra'). If you open the 'Rexx' logical window with awin , all the ARexx output will go to that logical window.

The front command brings the PowerVisor screen to the front.

Normally you would use 'rx' to start ARexx scripts. But you can also use this command to start ARexx commands from within PowerVisor :

```
< rx 'disp 3+4' <enter>
> 00000007 , 7
```

```
or something more useful :

< rx 'address command dir' <enter>
CLI> ...

And the directory appears on the shell output window. You can thus use
the 'rx' command to start cli commands from within PowerVisor.

If you have a pointer to a string in PowerVisor and you want to convert
this pointer to a real ARexx string you can use the 'string' command to
do that.


The last extra command for ARexx is  assign . With this command you
can assign something to a PowerVisor variable.
For example, the following ARexx script :

      file< /* */
      file< address rexx_powervisor
      file< a=1
      file< assign 'a=2'
      file< disp a
      file< disp 'a'

Will have as output :

      < rx file <enter>
      > 00000001 , 1
      > 00000002 , 2

You should understand why
   a=1
is not the same as
   assign 'a=1'

The first command assigns 1 to the ARexx variable 'a'. This variable
is not directly accessable from within PowerVisor.

The second command assign 1 to the PowerVisor variable 'a'. This variable
is not directly accesable from within the ARexx script. You can ask the
value of PowerVisor variables with  void  :

   ARexx< options results
   ARexx< a=1
   ARexx< assign 'a=2'
   ARexx< 'void a'
   ARexx< var1=result
   ARexx< 'void' a
   ARexx< var2=result

Note ! var1 will be equal to 2, but var2 will be equal to 1. Can you
explain why?

s:pv/ShowAscii.pv, s:pv/PrintMode.pv and s:pv/PrintHist.pv are three ARexx
script examples that you can examine.
s:pv/Assem.pv is a small assembler. This is also a good example but it is
```

rather large.

1.7 Scripts : Machinelanguage scripts

The following explanation is rather technical. Most PowerVisor users will probably never write machinelanguage scripts.

Machinelanguage scripts or ML-scripts are normal AmigaDOS executable files. In practice these can be programmed in several languages (like C), but for naming conventions we call them ML-scripts.

You can execute ML-scripts the same way as normal PowerVisor scripts : with the script command. PowerVisor will automatically check if the script is a machinelanguage script by reading the first four bytes. AmigaDOS executable files always begin with the same four bytes.

Example :

< script execfile 10 <enter>

will execute the file 'execfile' and give 10 as the first argument to the machinelanguage script.

When a ml-script gets executed some registers contain predefined information :

a0	A pointer to the rest of the commandline (to '10' in the previous example)
a1	Is the pointer to the 'rc' variable. You can use this pointer to store results or to get some value. The 'rc' variable is a longword
a2	Is the pointer to the PVCallTable. This is a table with routines for you to use. See the The wizard corner chapter for more information about the PVCallTable and the pvcall command

You can put the returnvalue in d0. This returnvalue will be the result of the 'script' command.

If you are serious about writing ML-scripts in C or machinelanguage you should look in the PVDevelop subdirectory. This directory contains all include files and libraries useful for developing ML-scripts.

This is really all there is to machinelanguage scripts. But you should read the The wizard corner chapter if you really want to make more advanced ML-scripts. Source/SearchHist.asm is an example ML-script. This script can be installed on a key to provide a history search utility (Note that this is already done in the standard PowerVisor-startup file provided with this release of PowerVisor).
The Source subdirectory contains some other examples (in C and machinelanguage) for ML-scripts. All compiled and assembled forms of these scripts are in the s:pv subdirectory. The C examples are written for SAS/C (or Lattice). If you have another compiler (like Aztec-C) you will probably have to change some things. I'm not sure if you will be able to

use the stubs (in the PVDevelop directory) for non-SAS/C compilers. But since the source of the stubs is provided it should not be difficult to addapt them to your needs.

1.8 Scripts : Making ML-scripts resident

If you want you can make ML-scripts resident in memory. This is certainly a lot faster if you use diskdrives and if the script is big.

You can use the resident command to make ML-scripts resident. Here follows an example :

```
< _a={-resident s:pv/SearchHist} <enter>
< attach 'go \(_a)' 04c 2 e <enter>
< attach 'go \(_a)' 04c 1 e <enter>
< remvar _a <enter>
```

These four commands install the 'SearchHist' ML-script on the shift-arrow up key. When these four commands are executed you can search in the PowerVisor history buffer with this key (This sequence of commands can be found in the PowerVisor-startup file).

The first command ('resident') loads the ML-script into memory and assigns the pointer to the code to the '_a' variable (I assume here that you know how the group operator works. See the Expressions chapter if you don't know it).

Using the attach command we attach the go command to the shift-arrow up key (we make a macro). Note that when 'attach' parses the command string it will replace \(_a) with the contents of the variable '_a'. This is the pointer to the code of the ML-script. You may ask why we didn't use the variable instead of the value of the variable? This is because we can know remove the variable since the value of the variable is burned in the macro string.

The last command removes the variable.

The 'go' command works very analogous to the script command. The only difference is that the script must be located in memory instead of a file. The register conventions are the same.

Now you can use <shift>+<up> to search in the history buffer. The text at the left of the cursor in the stringgagdet will remain unchanged. The script will search all lines beginning with this text.

You can use unresident to remove a loaded file. All resident files are also automatically removed when PowerVisor quits.