TechnicalInfo.hyper

COLLABORATORS

	TITLE : TechnicalInfo.hyper	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 5, 2023	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

TechnicalInfo.hyper

1.1 Technical Information (Wed Jul 15 15:24:32 1992)

Contents:

Back to main contents

1.2 Technical Information : Introduction

This file contains some technical info and details about PowerVisor that can be important to people writing scripts. If you are unsure about some special feature you may find it here.

1.3 Technical Information : Commandline parsing

This section contains very detailed information about commandline parsing.

The following steps happen when a command string is parsed. This parsing happens whenever a command is entered or for commands that are executed as arguments for other commands (like for , to , with , ...).

- Skip spaces
- Check for the comment character (';' by default)
  If it is a comment, the rest of the commandline is ignored and the parsing is stopped (0 is returned)
- Checks if there is an output suppressor ('-' by default)
  If there is one, the output to the current logical window is disabled.

- Skip spaces
- See if the command is an alias and if it is, replace the command string with the alias string. All '[]' symbols in the alias string are replaced with the rest of the commandline after the original command
- Skip spaces
- Check for the comment character (';' by default)
  If it is a comment, the rest of the commandline is ignored and the parsing is stopped (0 is returned)
- Checks if there is an output suppressor ('-' by default)
  If there is one, the output to the current logical window is disabled
- Skip spaces
- See if there is a '{' (command group). If there is we start parsing each individual command in this group (seperated by ';') with the algorithm described here. Note that comments do not work in groups except when the comment appears in the alias string. After parsing each individual command in the group, we stop the current parsing (we return the result code of the last command in the group)
- If there is no '{' we check if the command is an assignment.
  If it is we perform the assignment and stop the parsing (we return the assigned value)
- If we are still here, we have a normal command
- Skip spaces
- See if the command exists. If it does not exist we generate a 'Syntax Error'
- If the command is empty we simply continue the last memory listing (or disassembly)
- If the first argument of the command is a '?' we get the commandline template and show it (0 is returned if successful)
- Finally we come to execute the command (the result of this command is returned)

The situation is a bit different for the command typed in on the commandline (the stringgadget). BEFORE the above algorithm is executed, the following additional steps are called :

- Set the starting page for the current logical window to 0. This means that we will get '-MORE-' when a full page of output has passed
- Skip spaces
- Check if there is a feedback suppress character in front of the line ('~' by default). If there is, we remove it from the commandline and skip the following step
- Feedback the command on the current logical window (show the command as it was typed in)
- Display the '-BUSY-' prompt
- Skip spaces
- Check if there is a quick-execute char ('\' by default) in front of the commandline. When there is one we skip the following step and skip the last step of this procedure
- Check if there is a 'Pre' command to execute. A 'Pre' command is a command that is executed everytime the user enters a command (See The wizard corner for more information). If there is a 'Pre' command it is executed (with the previous procedure for parsing)
- If the previous 'Pre' command failed (if it was executed) we stop further parsing
- Use the previous procedure for parsing to parse the rest of the commandline

- (This step is also skipped if the quick-execute char is present) Check if there is a 'Post' command to execute. A 'Post' command is a command that is executed AFTER execution of the commandline (See The wizard corner for more information). If there is a 'Post' command it is executed (with the previous procedure for parsing

From these two procedures you can draw some conclusions :

- Alias expansion is only done once. This means that you can't use an alias in an alias. However, when you use this second alias in a group operator, it will be converted. For example :

```
alias disp 'print []'
alias print 'disp []'
```

  will simply redefine 'print' to mean 'disp' and 'disp' to mean 'print'.

```
alias disp '{print []}'
alias print '{disp []}'
```

  will cause an infinite loop when you call any of these two commands. This is because alias expansion is done again for each command in a group. PowerVisor checks for stackoverflow so don't worry.

- If you want more prefix operators on the same commandline you have to observe the following order :

```
~ \ -
```

1.4 Technical Information : Hold mode

While in hold mode, PowerVisor only waits for the hot key signal and for a crash. All other signals are ignored until the window is back again.

Starting with version 1.15ß, PowerVisor also checks for ARexx signals. So you can use the ARexx port while PowerVisor is in hold mode. However, there is currently one problem: when PowerVisor was on it's own screen before hold mode was selected (with the hold command), there may be a crash when you return from hold mode with the front command used from within ARexx. You can safely use this feature if the PowerVisor window is on the Workbench for example. This bug will be solved as soon as I can find it.

1.5 Technical Information : Expression parsing

Default operator priorities (you can change them with the pvcall command). Note that these priorities are the same as used in standard C.

Priorities between 1 and 10 are supported (1 is low priority)

Op	Function	Default priority

```
   -------------------------------------------
   *       Multiply            10
   /       Divide              10
   %       Modulo              10
   +       Add                 9
   -       Subtract            9
   <<      Left shift          8
   >>      Right shift         8
   >       Greater than        7
   <       Less than           7
   >=      Greater or equal    7
   <=      Less or equal       7
   !=      Not equal           6
   ==      Equal               6
   &       And                 5
   ^       Xor                 4
   |       Or                  3
   &&      Logical and         2
   ||      Logical or          1
```

Here are some remarks for expression evaluation :

- Decimal integers always start with a non-zero digit
- Hexadecimal integers start with a zero digit or with '$'
- String pointers always start with double quotes '"'
- '-','~' and '!' are unary operators. The element directly after the operator is evaluated and the operator is applied. This means that these unary operators have the highest priority possible. Unary operators are ALWAYS evaluated first.
- The '*' operator is also a unary operator. The same comments as for the normal unary operators apply here. The only difference is the dot '.' that may appear after the evaluated element.
- The ':' operator expects two string arguments (these string arguments do not support the '\' operator). The first argument (before the ':') is the list name. If you omit this argument the current list is used. The second argument (after the ':') is the list element. If you omit this argument the first element in the list with a name is used.
- The '&' operator. After the '&' operator follows a ':' operator.
- The '@' operator. The following '@' elements are supported :
  - d0 ... d7
  - a0 ... a6
  - sp
  - pc
- The '#' operator. After the '#' operator follows an integer (line number)
- A group operator '{' is evaluated by executing all commands in the group. The result of the group is the result of the last command in the group. The group operator is disabled when you use the expression evaluator from within a debug task.
- A normal string starting with a single quote '''. The following steps are used to evaluate a string starting with a quote :
  - The first thing to do is to check if the string is a symbol in the current debug task. This check is case sensitive.

```
            No abbreviations are possible.
         -  If the previous check failed we check if the string is an
            abbreviation for an element in the current list.
            This check is case insensitive.
         -  Otherwise an error is generated
   -  A name. This is a sequence of characters starting with a alphanumeric
      character or an underscore '_'.
      The following steps are used to evaluate the name :
         -  First we check if the name is a variable or a function.
            This check is case insensitive. No abbreviations are possible.
         -  If the previous check failed we check if the name is a
            loaded library function (with  loadfd ). This check is case
            insensitive. No abbreviations are possible.
         -  Otherwise we continue with the evaluation for strings with
            a single quote.


Here follows the syntax for PowerVisor expressions :

   <name> is a syntaxical element
   [] indicate optional items
   {} repeat the items 1 or more times
   |  choose between several items
   () group items
   #  is used for the ascii value of a character
   \  all items after this operator are excluded from the list
      before this operator (the 'all-except' operator)
   .. indicate a range


   <expression> ::= <element> [<operator> <expression>]
   <element> ::= <integer> | <unary operator> | <brackets> |
            <list operator> | <list address operator> | <group> |
            <line number operator>

   <integer> ::= <hexadecimal int> | <decimal int> | <variable> |
            <function> | <string pointer> | <debug symbol> |
            <current list element> | <register> | <library function>

   <hexadecimal int> ::= ('0' | '$') [{<hex-digit>}]
   <decimal int> ::= ('1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9') [{<dec-digit>}]

   <variable> ::= <name>

   <function> ::= <name> '(' [ <expression>
            [{<whitespace> <expression> }]] ')'

   <debug symbol> ::= <name> | <string>

   <current list element> ::= <name> | <string>

   <library function> ::= <name> '(' [ <expression>
            [{<whitespace> <expression> }]] ')'

   <register> ::= '@' ( ('d'|'D')(<dec-digit> \ ('8'|'9')) |
            ('a'|'A')(<dec-digit> \ ('8'|'9')) | 'sp' | 'pc' | 'sr')
```

```
<unary operator> ::= <negation> | <logical not> | <bitwise not> |
         <address operator>
<negation> ::= '-' <element>
<logical not> ::= '!' <element>
<bitwise not> ::= '~' <element>
<address operator> ::= '*' <element> ['.' ('b'|'w'|'l')]

<line number operator> ::= '#' <element>

<list operator> ::= [<name>] ':' [<name> | <string>]
<list address operator> ::= '&' <list operator>

<group> ::= '{' [{<command>}] ['}']
<command> ::= [<prefix operator>] ((<name> [{<expression>|<string>|
         <name>}]) | <assignment>)
<prefix operator> ::= '-' | ';'
<assignment> ::= (<name> | <register> | <address operator>) '='
         <expression>

<brackets> ::= '(' <expression> [')']

<name> ::= ('_'|<alphachar>) [{'_'|<alphachar>|<dec-digit>}]
<string> ::= ''' [{<string-char>|<quote operator>|<strong quote>}] [''']
<string pointer> ::= '"' [{<stringptr-char>|<quote operator>|
                     <strong quote>}] ['"']
<quote operator> ::= '\' (<integer quote>|<char quote>)
<strong quote> ::= '·' <endchar> {#<x>} <endchar>
<char quote> ::= <hex-digit> <hex-digit>
<integer quote> ::= '(' <expression> [<whitespace> <formatstring>] ')'
<formatstring> ::= {#<x> \ ')'}

<alphachar> ::= 'a'|'A'|'b'|'B'|...|'z'|'Z'
<hex-digit> ::= <dec-digit>|'a'|'b'|'c'|'d'|'e'|'f'|'A'|'B'|'C'|'D'|
         'E'|'F'
<dec-digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
<whitespace> ::= ' '|','|#9
<stringptr-char> ::= #<x> \ ('\'|'"')
<string-char> ::= #<x> \ ('\'|''')
<x> ::= 1..255
```