Expressions.hyper

COLLABORATORS			
	TITLE : Expressions.hyper		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		December 6, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Expressions.hyper

1.1 Expressions (Mon Nov 2 20:00:22 1992)

1.2 Expressions : Commands used in this tutorial

```
   disp        Display integer
   exec        Go to the ExecBase list
   help        Ask for help
   list        Show a list (tasks, libraries, message ports, ...)
   loadfd      Load a fd file
   memory      List memory
   print       Print a string
   remvar      Remove a variable
   scan        Ask input
   task        Go to the task list
   void        Evaluate the arguments given
```

1.3 Expressions : Functions used in this tutorial

alloc	Allocate memory
cols	Ask the number of columns in logical window
eval	Evaluate string
free	Free memory
if	Conditional evaluation
lines	Ask the number of rows in logical window

1.4 Expressions : Introduction

PowerVisor has a very powerful expression evaluator. Before you continue reading this file make sure you have mastered the basic features of PowerVisor (read Getting Started first).

First start PowerVisor (if it is not already in memory).

PowerVisor has two basic types: strings and integers. Sometimes these two types collide a bit. In this tutor file I will make you more comfortable with all the features PowerVisor has in this regard.

All commands and functions in PowerVisor expect integers and/or strings as their arguments. You can use expressions everywhere.

1.5 Expressions : Simple integers

First we have decimal integers :

```
< disp 5 <enter>
> 00000005 , 5

< disp 1236 <enter>
> 000004D4 , 1236

< disp -100 <enter>
> FFFFFF9C , -100
```

But not (can you see why ?) :

```
< disp 01236 <enter>
> 00001236 , 4662
```

This last notation is used for hexadecimal numbers :

```
< disp 01A <enter>
> 0000001A , 26

< disp $1a <enter>
> 0000001A , 26
```

You can see that PowerVisor has two ways to notate a hexadecimal number. PowerVisor interpretes '0<number>' as a hexadecimal number because this is

very convenient if you 'snap' (see Snapping away) a hex number from the
PowerVisor screen. Such a number almost always contains some leading
zeroes.

Note that in the current version of PowerVisor the following command :

```
< disp 1A <enter>
```

does not give an error :

```
> 00000001 , 1
```

This is in fact a bug and it will be solved in a later version of
PowerVisor.

1.6 Expressions : Expressions

You can use more complex expressions :

```
< disp 5+5 <enter>
> 0000000A , 10

< disp 1+(5*9)-(3&5) <enter>
> 0000002D , 45
```

Note that you cannot use spaces in expressions.

You can use the following binary operators. The list is from high to
low priority :

```
  * / %       multiply, integer divide, remainder
  + -         add, subtract
  << >>       left shift, right shift
  > < >= <=   integer comparisons
  != ==       not equal to, equal to
  &           bitwise and
  ^           bitwise xor
  |           bitwise or
  &&          logical and
  ||          logical or
```

There are also some unary operators :

```
  -           negation
  !           logical not
  ~           bitwise not
  *           contents operator
  &           address operator (explained later)
  :           list operator (explained later)
  @           special operator (for debugging)
  #           linenumber operator (for debugging)
```

You can also use brackets.
Some examples :

```
< disp 5+(9-(7*(5/(3+~1)))) <enter>
```

```
> FFFFFFEB , -21
```

Note that you MUST close all brackets.

The contents operator needs some more examples :

```
< disp *4 <enter>
> 07E007E4 , 132122596
```

(This is the pointer to execbase)

```
< disp *4.b <enter>
> 00000007 , 7

< disp *4.w <enter>
> 000007E0 , 2016

< disp *(2+2).l <enter>
> 07E007E4 , 132122596
```

The syntax of the * operator is:

```
   '*'<expression>['.' ('b' | 'w' | 'l') ]
```

'b', 'w' and 'l' are the size indicators (b = byte, w = word, l = long). If you do not specify '.', PowerVisor assumes long mode. This operator checks for illegal addresses :

```
< disp *5 <enter>
> Odd address error !
```

On the 68000 processor you cannot read a long at an odd address. PowerVisor will always give this error even if you have a 68020/68030 processor.

1.7 Expressions : String pointers

A string pointer is NOT a string, it is an integer. When you use a string pointer, PowerVisor will allocate a temporary space for this string and give you the address.

String pointers (and strings for that matter) support the following operators :

```
   \           the quote operator
                     \<hex digit><hex digit>
                        is replaced by the ascii character
                     \(<expression>)
                        is replaced by the string representation
                        of the <expression>
                     \(<expression>,<formatstring>)
                        is replaced by <expression> formatted like
                        <formatstring>. <formatstring> is a C-style
                        (printf) formatting string. Use %ld for
                        integers, %lx for hexadecimal integers, %s
                        for strings and %c for characters
                     \n
                        newline (equivalent to \0a)
```

```
                    \01
                       normal attributes for this line. The occurence
                       of this character in an output string causes the
                       current line to have the normal attributes
                       (colours)
                    \02
                       hilight the current line. The occurence of this
                       character in an output string causes the hilighting
                       of the current line
                    \03
                       inverse video for the current line. The occurence
                       of this character in an output string causes the
                       current line to be output in inverse video. Note
                       that inverse video and hilight are the same if
                       the PowerVisor screen is only one bitplane (nofancy
                       mode)
                    \04
                       hilight and inverse video together. If the
                       PowerVisor screen is only one bitplane these two
                       cancel each other, so this will be equivalent to
                       \01
                    \<anything else>
                       is simply equivalent to <anything else>. This
                       is useful for quotes (like " and ’) and for
                       the backslash \
   ·          the ’strong quote’ operator
                    ·<end char> various characters of all sorts <end char>
                       To type the dot use <alt>+8. Most users will
                       probably never need this operator. It is useful
                       in aliases if you want to take away ALL special
                       interpretations of characters. The ONLY thing
                       that will stop the parsing of the string is the
                       <end char>.
                       The  salias  alias is an example of an alias
                       using the strong quote.
```

Example :

```
< disp "Hello" <enter>
> 07E50E52 , 132451922

< memory 07E50E52 10 <enter>
> 07E50E52: 68656C6C 6F0007E2 1010                           Hello.....
```

PowerVisor remembers the last 10 strings and string pointers (all in one pool) before it frees them. This means that you can only use 10 string pointers in one command at the same time.
If you want a permanent string pointer you can use the alloc() function.

The next example illustrates the use of the contents operator (*) and a string pointer :

```
< disp *("Hello"+1).b <enter>
> 00000065 , 101
```

```
(101 is the ascii value for 'e').

Look at the following example :

< memory "Left\41\42\43Right" 16 <enter>
> 07EC98A2: 4C656674 41424352 69676874 000007EC            LeftABCRight....

The '\' notation is useful for unprintable characters (and untypable).
After the '\' follow two hexadecimal digits. If the first character
following the '\' is not a hexadecimal digit the first '\' is ignored :

< memory "Left\Right" 10 <enter>
> 07E5C0BA: 4C656674 5C526967 6874                         Left\Right

< memory "Left\"Right" 10 <enter>
> 07E5C0BA: 4C656674 22526967 6874                         Left"Right


There is one extra feature that you have with quoting. Consider the
following example :

< memory "Left\(4+5)Right" 16 <enter>
> 07E31A72: 4C656674 39526967 68740000 00000000            Left9Right......

The expression between the two brackets can be as complex as you wish.
You can use variables, functions, ...
You can use an optional format string directly after the expression
(include some white space of course). Use standard C-formatting
conventions :

< memory "Left(4+5,%02ld)Right" 16 <enter>
> 07E31A72: 4C656674 30395269 67687400 00000000            Left09Right.....

< memory "Left(65,%lc)Right" 16 <enter>
> 07E31A72: 4C656674 41526967 68740000 00000000            LeftARight......
```

1.8 Expressions : Names

```
Names are probably the most difficult things in the PowerVisor parsing
language. They can be almost everything. I think examples are the best
way to illustrate their purpose :

< task <enter>
< list <enter>

> Task node name      : Node     Pri   StackU   StackS Stat Command         Acc
> --------------------------------------------------------------------------------
> ConClip Process     : 07E60410 00       242     4000 Wait sys:c/ConCl(02) -
> RexxMaster          : 07E6AA48 04       162     2048 Wait            (00) -
> « IPrefs »          : 07E59568 00       862     3500 Wait            PROC -
> ClickToFront        : 07E75210 15       398     4096 Wait            PROC -
> PowerSnap 1.1 by Nic: 07E9B630 05       102     2000 Wait            PROC -
> PowerVisor.task     : 07E7B4F0 00        82     4096 Wait            TASK -
> WB_2.x              : 07E0FF38 0A       130     2400 Wait            PROC -
> DF0                 : 07E15910 0A       130     2400 Wait            PROC -
> ...

< disp powervisor <enter>
> 07E7B4F0 , 132625648
```

In this case we used a name ('powervisor') as an abbreviation for an element in the current list. We could also have written :

```
< disp POWERvi <enter>
```

or

```
< disp 'Powervisor.' <enter>
```

or

```
< disp power <enter>
```

The last command is correct but it is ambiguous in this case since there are two names in the current list beginning with 'power'. When this is the case the first match is used (PowerSnap 1.0 in this case).

Important to remember is that when we use names for this purpose the following rules apply :

- you may use quotes (if you want to include spaces for example)
- the searching is case insensitive
- you can use abbreviations, i.e. you need not give the full name

Consider the following example :

```
< powervisor=5 <enter>

< disp powervisor <enter>
> 00000005 , 5

< disp 'powervisor' <enter>
> 07E7CBE0 , 132631520

< disp powerviso <enter>
> 07E7CBE0 , 132631520

< disp powervisor1 <enter>
> 07E7CBE0 , 132631520
```

We have created a variable 'powervisor' with value 5. The name 'powervisor' has now lost it's meaning as an abbreviation for an element in the current list. Note that we can still use 'powerviso' and 'powervisor.' for the element in the current list.
So we see that the following rules apply for variables :

- you can't use quotes for variable names (if you do it is interpreted as an element in the current list)
- variable names can't be abbreviated
- variable names are case insensitive

When you use quotes you force interpretation of the current list. There is a shorter and better way to do this (also see The list operator) :

```
< disp :powervisor <enter>
> 07E7CBE0 , 132631520
```

Now we remove the created variable :

```
< remvar powervisor <enter>
```

There is still a third way to interprete names. But it is at this moment not appropriate to give examples. A name can also be a symbol for the current debug task (see Debugging). The rules for symbol names are :

- you can use quotes
- symbol names are case sensitive
- no abbreviations are possible

In case of ambiguity observe the following order of checking :

- PowerVisor will first check if it is a variable
- If it is not a variable, PowerVisor will check if it is a library function (see later)
- If it is not a library function it could be a symbol for the current debug task
- If it is not a symbol PowerVisor will search the current list

If you use quotes for the name, PowerVisor will skip the variable testing and only test if it is a library function, a symbol or a list element. If you use the ':' operator (see above) PowerVisor will only search the current list.

Warning! A name is strictly a sequence of characters. If you want to use special operators like the quote operator you should normally use strings (with single quotes). In most cases PowerVisor will not complain when you use a quote operator or something else in a name, but remember that the result is not always satisfactory. Problems can occur when you use the name in complex expressions containing functions and groups. If you use the quote operator in that case, the chances are high that PowerVisor will get confused. You can get a error message like :
   'Your brackets are really out of order !'
even if your brackets may seem allright to you.

The following is an illegal example :

```
< a=alloc(s,' testing ') <enter>

< d if(1,{print test\(a,%s)it\n},2) <enter>
> Your brackets are really out of order !
```

You should type :

```
< d if(1,{print 'test\(a,%s)it\n'},2) <enter>
> test testing it
> 07EECDA8 , 133090728
```

1.9 Expressions : Functions

Functions are a special form of variables. Internally they are almost the
same. Type :

```
< help functions <enter>

> General functions
> -----------------
> ALLOC      : allocate memory          LASTMEM    : give last memory
> FREE       : free memory              LASTFOUND  : last search address
> REALLOC    : reallocate memory        PEEK       : peek value in structure
> GETSIZE    : give size of memoryblock APEEK      : peek address
> ISALLOC    : is memory a pv-block ?   STSIZE     : get structure size
> KEY        : returns pressed key      RFCMD      : refresh command
> QUAL       : qualifier for last key   RFRATE     : refresh rate
> GETCOL     : get logical col width    GETLWIN    : current logical window
> GETROW     : get logical row height   GETERROR   : get error of routine
> TOPPC      : get debug win top pc     TAGLIST    : get current tag list
> BOTPC      : get debug win bottom pc  EVAL       : evaluate argument string
> ISBREAK    : check if breakpoint      IF         : conditional evaluation
> GETDEBUG   : get current debug ptr    CURLIST    : current list
> GETX       : get the current x coord  COLS       : get max nr of cols
> GETY       : get y coord              LINES      : get max nr lines
> GETCHAR    : get the current char     BASE       : get first listelem
```

Since functions are so much like variables the same evaluation rules apply
to them (see Names). You must use the full name
(no abbreviations). The only exception is that you must use brackets after
the function name even if there are no arguments.

Some examples :

```
< disp lines(main)
> 00000035 , 53
```

This means that I have 53 lines on my PowerVisor screen.
(See lines() for more complete information about this function).
'main' is the argument for the function.

The following is incorrect (There is a space between 'lines' and
'(main)') :

```
< disp lines (main)
> You must use brackets with functions !
```

1.10 Expressions : Library functions

PowerVisor has the very powerful capability to execute library functions.
You only need to load the corresponding fd-file (in the fd2.0 or fd1.3
subdirectory) :

```
< loadfd exec :fd2.0/exec_lib.fd <enter>
```

or

```
< loadfd libs:exec :fd2.0/exec_lib.fd <enter>

> New functions: 0000007E,126
```

PowerVisor will then know how to call all functions from the exec library.

You can now use all the exec library functions as if they were normal PowerVisor functions :

```
< disp typeofmem(100000) <enter>
> 00000303 , 771

< disp openlibrary("exec.library",0) <enter>
> 07E007E4 , 132122596
```

The following rules for library functions apply :

- you MUST use brackets even if there are no arguments (like functions)
- you MUST close the brackets for the library function arguments. This means that if you use an expression as an argument you must close the brackets for this expression as well
- you cannot use abbreviations for library functions
- library function names are case insensitive
- you can use quotes (for example to skip variable parsing, this is needed for the ’input’ library function in dos.library since there is a built-in ’input’ variable in PowerVisor)

1.11 Expressions : The list operator

Example :

```
< exec <enter>
< list <enter>

> SoftVer      : 00CF     | LowMemChkSum : 0000     | ChkBase      : F81FF81B
> ColdCapture  : 00000000 | CoolCapture  : 00000000 | WarmCapture  : 00000000
> SysStkUpper  : 07E02248 | SysStkLower  : 07E00A48 | MaxLocMem    : 00200000
> DebugEntry   : 00F82E28 | DebugData    : 00000000 | AlertData    : 00000000
> MaxExtMem    : 00000000 | ChkSum       : A366     | ThisTask     : 07E7B418
> IdleCount    : 000CC6E6 | DispCount    : 0005444A | Quantum      : 0004
> Elapsed      : 0004     | SysFlags     : 0000     | IDNestCnt    : FF
> TDNestCnt    : F4       | AttnFlags    : 0017     | AttnResched  : 0000
> ResModules   : 07E00428 | TaskTrapCode : 07E80AE6 | TaskExceptCod: 00F83A9C
> TaskExitCode : 00F823D0 | TaskSigAlloc : 0000FFFF | TaskTrapAlloc: 8000
> VBlankFreq   : 32       | PowerSupplyFr: 32       | KickTagPtr   : 00000000
> KickCheckSum : 00000000 | RamLibPrivate: 07E1F470 | EClockFreq   : 000AD303
> CacheCtrl    : 00002919 | TaskID       : 00000001 | PuddleSize   : 00000000
> MMULock      : 00000000 |
```

We are now in the execbase structure list.

```
< disp quantum <enter>
> 00000004 , 4
```

The same rules apply as for normal list element searching (see the Names section).

If there are possible ambiguities you can use the ':' operator.

```
< disp :quantum <enter>
> 00000004 , 4
```

If you want to change the quantum variable you can use the '&' operator (This operator is only supported for lists of this type (exec, graf, intb, ...)) :

```
< disp &exec:quantum <enter>
> 07E00904 , 132122884
```

This operator returns the address of the quantum variable in the exec base list.

You can now use this address to change the variable (also see Assignment) :

```
< *&exec:quantum.w=16 <enter>
```

You can also use the list name before the operator (useful if you are in another current list) :

```
< d exec:quantum <enter>
> 00000004 , 4
```

1.12 Expressions : Assignment

We have already used assignment a few times. We used it to assign a value to a variable, and we used it (in the previous section) to assign a value to a memory location. Here are some more examples :

```
< a=4 <enter>
< b=5 <enter>
< disp a+b <enter>
> 00000009 , 9

< var100=100 <enter>
< var100=var100+var100 <enter>
< disp var100+var100 <enter>
> 00000190 , 400
```

In this form we use assignment to assign a value to a variable.

You may put spaces round the '=' symbol (but remember that in general you still can't uses spaces in expressions) like this :

```
< a = 4 <enter>
```

One more example :

```
< mem=alloc(n,100) <enter>
< *mem=$11111111 <enter>
< *(mem+4).w=$2222 <enter>
< *(mem+6).b=$33 <enter>

< memory mem 16 <enter>
> 07E7761A: 11111111  22223300  00000000  00000000           ....""3.........

< void free(mem) <enter>
```

With the alloc() function we allocated 100 bytes of memory. (The ’n’ argument means the next argument is a number. Then we fill this memory with some values. With the memory command we display them.
After that we use the free() function to free our allocated memory. Note that PowerVisor will automatically free memory allocated with ’alloc’ when PowerVisor quits.

1.13 Expressions : The group operator

Sometimes it is conveniant to group several commands together. You can do just this with the group operator. You can use the group operator in two different ways. As a command or as a function. Here are some examples :

Using the group operator as a command :

```
< {disp 3;disp 5;disp 7} <enter>
> 00000003 , 3
> 00000005 , 5
> 00000007 , 7

< {{disp 1;{{disp 2};disp 3};disp 4};disp 5} <enter>
> 00000001 , 1
> 00000002 , 2
> 00000003 , 3
> 00000004 , 4
> 00000005 , 5

< {a=2;b=3;disp a*b} <enter>
> 00000006 , 6
```

You can see that the sole purpose of this operator is to allow you to give more than one command on the commandline. This can be very useful when some command expects another command as an argument (for example the refresh command).

You can also use the group operator as a function :

```
< disp {a=4}*2 <enter>
> 00000008 , 8
```

This command will first assign 4 to ’a’ and then execute ’disp 4*2’.

```
< disp {a=4;temp=a*7;void temp+temp*(temp/4)} <enter>
> 000000E0 , 224
```

The void command simply evaluates all it’s arguments. It does not display anything.

```
< disp {disp 1}+2 <enter>
> 00000001 , 1
> 00000003 , 3
```

When the group operator is used as a function, it is the last executed command in this group that determines the return value. The return value is different for each command. Look at Command Reference for more details.

1.14 Expressions : Strings

All previous sections covered integers. PowerVisor also uses strings. A string is very easy. When a command expects a string, almost everything is correct. Some examples :

```
< print Hello <enter>
> Hello

< print Hello\0a <enter>
> Hello
```

Notice the difference between these two commands. If you do not explicitely ask for a <enter> at the end of the line you will not get one. ’\0a’ is the linefeed character (you can also use ’\n’).

```
< print ’Hello\n’ <enter>
> Hello

< print 5+6 <enter>
> 5+6

< print Complete rubbish <enter>
> Complete

< print ’Complete rubbish’ <enter>
> Complete rubbish

< print ’Complete rubbish <enter>
> Complete rubbish

< print "Complete rubbish" <enter>
> Complete rubbish

< print ’a\nb\nc\nd\n’ <enter>
> a
> b
> c
> d

< print Hello\ you <enter>
> Hello you
```

```
< print 'Hello you' <enter>
> Hello you
```

Strings are very versatile. Even the notation for string pointers (double quotes) is accepted.

You can also use the backquote operator :

```
< a="Hello there\n" <enter>
< print a <enter>
> a

< print \(a,%s)\n <enter>
> Hello there
```

Strings (like string pointers) also support the special integer quoting feature :

```
< a=1 <enter>
< print 'Testing \(a) \(2,%03ld) \(11+11+11,%04lx) \(65,%lc) !\n' <enter>
> Testing 1 002 0021 A !

< print 'First string : \("Second string",%6.6s).\n' <enter>
> First string : Second.
```

1.15 Expressions : Expressions and debugging

If you are debugging a program (see the Debugging chapter for more information) you can also ask the contents of a register with the '@' operator. The following registers are possible : @d0..@d7, @a0..@a6, @sp, In addition to the '@' operator you can also use the '#' operator. This operator returns the address of the given linenumber in the currently loaded source. You can only use this operator when you are debugging.

1.16 Expressions : Some useful functions

You can make conditional expressions using the if() function. Here is an example :

```
< a=2 <enter>
< disp if(a==2,1000+5,2000) <enter>
> 000003ED , 1005
```

A very complex example :

```
< disp if(a==3,{disp 3*3;void 1},if(a==2,{disp 2*2;void 1},0))
> 00000004 , 4
> 00000001 , 1
```

This completely useless command computes the square of the variable 'a', but only if 'a' is equal to 2 or to 3. It also prints the value 1 if 'a' is equal to 2 or to 3, and 0 if it isn't.

```
I think some detailed explanation can be useful here :-)

   The  disp  command takes it's first argument and prints it.
   This argument is equal to :

      if(a==3,{disp 3*3;void 1},if(a==2,{disp 2*2;void 1},0))

   The first thing that is evaluated is the 'if' function. The 'if'
   function has three arguments :

      one number :
         a==3

      and two other expressions :
         {disp 3*3;void 1}
         if(a==2,{disp 2*2;void 1},0)

   If 'a' is equal to 3 the first string is taken and evaluated.

      This results in evaluation of the following expression :

         {disp 3*3;void 1}

      This is a group expression (see  The group operator ). The group
      operator executes all the commands in it and returns as a result the
      result from the last executed command. This results in the execution
      of :

         disp 3*3

      and

         void 1

      So 9 is printed on the screen and 1 is returned as a result from
      the group operator (the  void  command simply evaluates all it's
      arguments).

      So the result of the first 'if' function is 1 (but 9 is already
      printed). This result is printed. So you have 9 and 1 as output.

      ! End evaluation !

   If 'a' is not equal to 3 the second string is taken and evaluated.

      This results in evaluation of the following string :

         if(a==2,{disp 2*2;void 1},0)

      This is again an 'if' expression and is evaluated analogous.

If you happen to have an expression in a string (this could be a string
typed in by the user) you can evaluate it using the  eval()  function :

Ask for input ( scan  returns a pointer to a string. We can also find this
pointer in the predefined constant 'input') :
```

```
< scan <enter>
```

Type an expression :

```
????< 10+5 <enter>
```

See if the string is really correct :

```
< print \(input,%s)\n <enter>
> 10+5
```

Evaluate it :

```
< disp eval(input) <enter>
> 0000000F , 15
```

or you can of course type :

```
< disp eval("10+5") <enter>
> 0000000F , 15
```