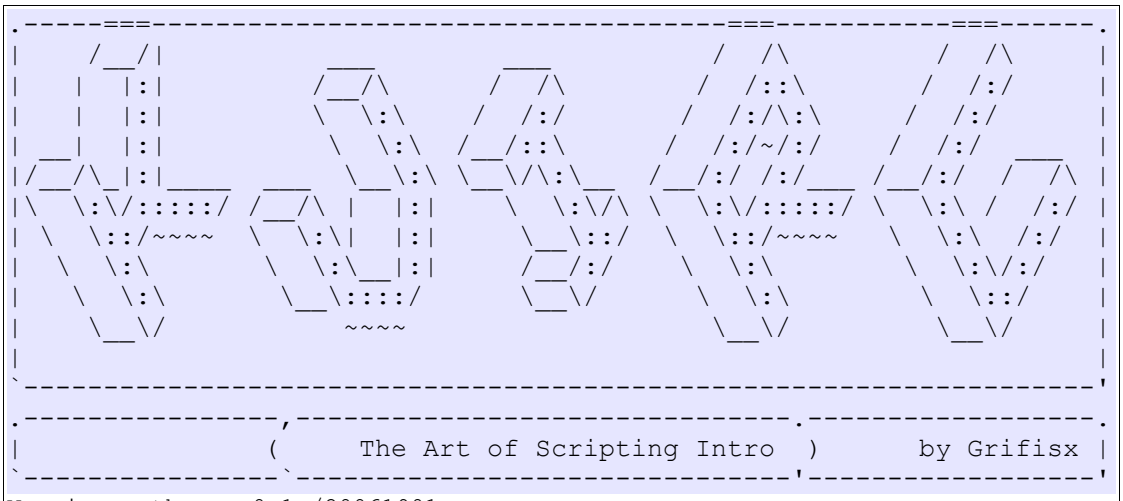

Version: etherea-0.1a/20061901

A brief introduction to KVIrc scripting language

Start:

This is meant to be such a Start about programming, a summary of functions, commands and some of the most important syntaxes.

Translated and adapted from the original handbook (so tnx to Pragma =D).

Index:

Variables

Global Variables:

The name of a Global Variable is defined through the percentage symbol (%), followed by a capital letter starting from 'A' to 'Z', after followed by a train of letters (lowercased or capitalized) or numbers or symbols (from 'a' to 'z', and from '0' to '9','.','_').

Example:
"%INDEX","%My_nickname","%Index","%Article" and "%article_number", a Global Variable does exists for the whole application' life.

Try this out:
/%Hello = "Hello world!"
Now from any window, or in such whatever script, the variable %Hello will exists with the content: "Hello world!".

Try in any window:
/echo %Hello
so you can face it.

Local Variables:

The name of a Local Variable is defined by the percentage symbol (%), followed by a lowercase letter starting from 'a' to 'z', and followed by a train of letters, numbers or symbols ('.','_').

Example:
"%index","%my_nickname","%foo","%bAR1" and "%foo.BAR"

A local variable is shown inside the instruction block where it is created, for example, in an alias, or inside an existing code.

Try this out:
/%hello = "Hello world!"

And now from any window try:
echo %hello

The result will be... nothing! That's just because the life of the variable has been terminated with the command execution.
Variables are created with the assignment of an own value and are destroyed with the assignment of a null value, for example:

Creation:
%nick=”Grifisx”
Destruction:
%nick=””

Array:

An array is a collection of variable data indexed by number; the first index of an array is 0 while the last is equal to the lenght minus one (so that counting starts from zero).
To obtain the exact number of elements in an array we can use this expression: *%ArrayExample[]#* (with upcomed version starting from “Anomalies” to furthers using even the *$lenght(arrayname[])* that is more suitable obtaing a more clear and readable code).

It is not necessary to declare the lenght of the array as in many other programming languages, once a number has come to be added, the lenght will vary automatically; if the first assigned element will be instructed to an index higher than 0 (>0), all the previous positions will be empty.

For example:

```
%Array[0]=Grifisx
%Array[1]=Noldor
%Array[2]=Pragma
#Print all the content of the array
echo %Array[]
#Print the lenght of the array
echo %Array[]#
echo $length(%Array[])
#Print only the first element
echo %Array[0]
```

Just put it in the Tester Script and execute.

And now try out this code:

```
%Array[0]=Grifisx
%Array[1]=Do not show this
%Array[2]=Noldor
%Array[5]=Secret shhhh..
%Array[8]=Pragma
for(%i=0;%i < $length(%Array[]);%i+=2)echo Entry %i: \"%Array[%i]\";
```

As you can see it is really simple to create collections indexed by numbers, so it's the easier to move inside them; here was the aim to use a *for cycle* but same could be using a *foreach(%item,%Array[])echo %item*, or a *while* cycle.
An array can be initialized in this way:
%Array[]=$array(Grifisx,Noldor,Pragma,etherea);
that is using the function *$array(<el1>,<el2>,<el3>,<el4>,..).*

And cames now the time to have a look to the older brother of array: the dictionary.

Dictionary

Dictionaries are associative array with string as values (easy said an array with not numeric index); and let's have a closer look to the example:

```
%Songs{Jimi Hendrix} = Voodo child
%Songs{Shawn Lane} = Gray piano's flying
%Songs{Imogen Heap} = Hide and Seek
%Songs{Greg Howe} = "Full Throttle"
# Show everything in a string
echo %Songs{}
# Show every element of the dictionary
foreach(%var,%Songs{})echo %var
```

Naturally, in here such as in arrays, *%Songs{}#* will give back the number of the elements of the dictionary.
While *%Songs{}@* will give back a list of those same elements separated by commas.

Operators and Assignment

"="(Assignment)

Assignment operator is "=" and work as for very other language, see some examples:

```
# Assignment to Local Variable %idx the value of 0
%idx=0;
# Assignment to Global Variable %My_Nick my nickname
%My_Nick = "Grifisx";
# Assignment to Variable "%name" the value gained from a function
%name = $function();
# Recording in element 0 of the array "%Addresses" the string "start"
%Addresses[0]="start";
```

"=~"(Binding) ***advanced scripting***

This operator is for a more experienced user handling.
It's useful to have searches and substitutions inside a string, even using regular expressions.

Basic syntax:
<basic_string> =~ <operation>**[parameters]**

Where *<operation>* can be 't','s'.
<basic_string> is the string on which get operation *<operation>*.
-'t' is used for replacing letters.

The complete syntax is:
<basic_string>=~t/<ToBeReplacedChars>/<ReplacedChars>/

This operation can even be done with 's' or 'tr' (to preserve languages compatibility).

For Example:

```
%A=This is a test string
echo %A
%A=~ tr/abcdefghi/ABCDEFGHI/
echo %A
```

-'s' is used to replace letters associations.

The complete syntax is then:
<basic_string=~s/<pattern_ToSearchFor>/<pattern_Replaced>/[flags]

Here's an example with regular expression:

```
%A=This is a test string
echo %A
%A=~ s/([a-z])i([a-z])/\\1I\\2/
echo %A
%A=~ s/([a-z])i([a-z])/\\1@\\2/gi
echo %A
```

[flags] can be a combo of letters 'g','i' e 'w'.
'g' ->globally performs a search skipping the first stop on first *<pattern_ToSearchFor>* occurrency.
'i' ->performs a non case sensitive search.
'w' ->performs a simple wildcards search.

"X=" Arithmetic operations

General syntax:
<left_operand> <operation> <right_operand>

Where *<left_operand>* and *<right_operand>* must be numbers.

All these operations do *<operation>* (formerly be +,-,*,/,%,|,&) between the right operator and the left operator, so that the result is kept in the left operator (that should be a variable or an element of an array or an element of a dictionary).

<operation> can be:
+= : sums *<right_operand>* to *<left_operand>*
-= : substracts *<right_operand>* from *<left_operand>*
*= : multiplies *<left_operand>* by *<right_operand>*
%= : computes the module *<right_operand>* by *<left_operand>*
|= : computes logical OR between *<left_operand>* and *<right_operand>*
&= : computes logical AND between *<left_operand>* and *<right_operand>*
/= : divides *<left_operand>* in *<right_operand>*

For example:
```
%A=8
%A+=3
echo %A
```

"++,--" Operators for increment and decrement

These two operators are only working with numeric variables.

General syntax:
<left_operand> <operator>
++ increments *<left_operand>* by a single unit
-- decrements *<left_operand>* by a single unit

And this turns out to be *+= 1* or *-= 1*.

For example:
```
%A=3
%A++
echo %A
```

The result is obviously **4**.

These operators are often used in cycles like:
```
%idx=0;
while(%idx==8)
{
      echo Actual Value %idx;
      %idx++;
}
```

"<<,<+,<" Operators of strings interlink

Operator <+ : appends <*right_operand*> to <*left_operand*> continually
Operator << : appends <*right_operand*> to <*left_operand*> unlinking with a blank space
Operator <, : similar to '<<' appends dividing operands with a ','.

For example:
```
%sent=Hello
%sent <, whassup
%sent << ?
echo %sent
```

Let's put it in the Script Tester and sit down watching those results.

Cycles

while

Syntax:
while(<condition>){<commands>};

The while command executes in a very looping way a command or a block of <*commands*> until it verifies (or do not verifies) the given <*condition*>.

For example:
```
%i = 0;
while(%i != 10)
{
      echo %i;
      %i++;
}
echo End
```

Here's an example with a single command:
```
%i = 0;
while(%i != 10) %i++;
echo %i;
```

Cycle can be interrupted with the *break* command.

Comparison methods that can be used are:
== (is equal to)
!= (is different from)
<= (is lower or equal to)
>= (is greater or equal to)
< (is lower than)
> (is greater than)
!<*variable*> (that doesn't exist, as for example: *if(!%nick){<commands}* means there is no variable %nick so it has no value)
<*variabile*> (that does exists, as for example Es: *if(%nick){}* means if there a variable %nick it has a value).

It is also possible to use multiple conditions chaining with the **&&** (correspondent to the logic AND, so that commands are executed only if both given conditions are true) or with **||** (correspondent to the

logical OR, so that commands are executed only if almost one of the given conditions is true).

for

Sintax:
for([initialization];<condition>;[operation]){<commands>;}

The *for(;;){}* cycle allows the initialization of the variable used as index to loop, sets the valid condition to execute the loop itself and the operation granting to act on the index, all in a complex but single command.

This cycle can be break with the command *break.*

For example:

```
# With a single command
for(%i = 0;%i < 100;%i++)echo %i
for(%i = 100;%i;%i -= 10)echo %i
%i = 0
# With a block of commands
for(;%i;)
{
      echo %i
      %i++
      if(%i > 10)
      break
}
```

do

Sintax:
do {<*command*>} while (<*condition*>)

Executes <*command*> once then evaluates the <*condition*>.
If <*condition*> evaluates to true (non zero result) then repeats the execution again.
<*command*> may be either a single command or a block of commands.

For example:

```
%i = 0;
do %i++; while(%i < 100);
echo "After first execution:  %i";
%i = 10
do {
      echo "Executed!";
      %i++;
}
 while(%i < 1)
echo "After second execution:  %i";
```

foreach

Sintax:
foreach(<*variable*>,[<item>[,<item2>[,<item3>[.....]]]){<*command*>};

It executes the <command> (also a block of commands closed within {} as for the *while* and for cycle), until it can assign an *<itemX>* to *<variable>*.
<item> can also be a simple variable, an array, a dictionary or a function giving back a value or a list of values.

For example:
```
foreach(%i,1,2,3,4,5,6,7,8,9)echo %i
```

Or just try in a channel window:
/foreach(%nick,$chan.users) echo User: %nick

In this last case the result is being justified by the fact the function *$chan.users* gives back the list of all nicknames currently on a channel, separated by a comma.

Try this out:
/echo $chan.users.

Control conditions

if/else

Sintax:
if(<*condition*>){<*command1*>}[else {<*command2*>}]
It executes the *<command1>* (or the block of commands) if the *<condition>* is true; if used, *else* is going to execute the *<command2>* (or the second block of command) if the *if* *<condition>* is false.

For example:
```
%idx=0
while(%idx<=10)
{
      if(%idx==3 && %idx!=6) echo THREE
            else echo %idx
      if(%idx==6)
      {
            echo S
            echo I
            echo X
      }
      else
      {
            echo ............
      }
%idx++
}
```

In this loop too, as in the *while* cycle, confrontation methods can be used and defined as follows:

== (is equal to)
!= (is different/not equal from)
<= (is lower or equal to)

>= (is greater or equal to)
< (is lower than)
> (is greater than)
!<variable> (it doesn't exists, as for example *if(!%nick){<commands}* means if there is not the variable %nick it has no value)
<variabile> (it does exists, as for example *if(%nick){}* means if there is the variable %nick it has a value).

There can even be the chance to use multiple conditions interlinking with the "**&&**" (corresponds to the logic AND and commands are executed only if both given conditions are true) or with "**||**" (correspnds to the logical OR, commands are executed only if one of the cgiven conditions is true).

switch\case

Sintax:

```
switch [-s] (<expression>)
{
      case(<value>)[:]<command>
      [break]
      match(<wildcard_expression>)[:]<command>
      [break]
      regexpr(<regular_expression>)[:]<command>
      [break]
      default[:]<command>
      [break]
}
```

The *switch* construct has been enriched (on the way of the normal construct found in the C language) by two new labels: *match()* and *case.*

match() allows the comparison through simple wildcards:

```
%nick=Grifisx
switch(%nick)
{
      match(*r?fisx)
      {
            echo Hello Grifisx
            break
      }
      match(*W?fisx)
      {
            echo who are you?
            break
      }

}
```

And then *case:*

```
%nick=Grifisx
switch(%nick)
{
      case(Grifisx)
      {
            echo Hello Grifisx
            break
```

```
      }
      case(WHO)
      {
            echo who are you?
            break
      }

}
```

And the label *regexpr* that allows more experienced users to use and compare even using regular expressions.

switch (-s) canbe used to allow comparison in non sensitive cases (distinguishing lower or capital letters).

/ECHO STOP.

- - - - - - -- - - - - - - - -- - - - - - - - - -- - - - - -- - - - - -
"You see things; and you say `*Why?*' But I dream things that never were; and I say `*Why not?*"
(George Bernad Shaw)
- - - - - - - - -- - - - - - - - -- -- - - - - - - - - - - - - - - - -
Grifisx