Table of contents

ELF

A Client for playing MUDs

Written by Alfredo Milani-Comparetti © 1995-96

Many thanks to all those who helped me writing this piece of software, unconditionally testing it and morally supporting its development, since when it appeared to be simply a dream till now that it appears in the final release!
Thanks to Giovanni Gargani (Thanatos) for its DALE, the program that introduced me to the world of MUDs.
A very special thank to Agostino Fanti (Pindus) for his endless suggestions, for his faith in the good working of the darkest version of this program, for his patience in tolerating my typical programmer's strange points of view, and for his being... himself :-)
Many many thanks to those who helped ELF spreading in the world :-)

Basic concepts

ELF is a program to play MUDs. The basic idea was given by DALE, a program for playing DikuMUD's written by Giovanni Gargani.

My wish to write a completely mine program, as well as the need to be able to add anything I would have ever liked to, brought ELF to the light.

ELF comes to you with a full package of triggers and a list of MOBs and OBJECTs with the corresponding translation (The Ghost of Brackenred=bones).

To make a long story short: use ELF and let me know what you like or not.

MAIN FEATURES:

- full ANSI colours support
- command line editing with history, insert e delete
- buffered and unbuffered operating mode
- SCROLL-BUFFER with colours
- macros, triggers, variables and aliases
- easy translation of long names
- multiple commands on the same line
- user-friendly on-line editing of macros, triggers, variables and aliases
- the ability to enable/disable each single trigger
- the ability to automatically recognise the length of the tick, showing when it will end
- user-defined variables inside macros, triggers, and commands
- file handling commands and functions
- special meta-commands like @IF, @SKIP, @BEEP, @CALL, @WHILE and @OUTSTR
- expression evaluation with functions like @TIME, @TICK, @ONLINE and @IDLE
- functions @CRY and @DCRY to encrypt and decrypt some text using a key
- advanced programming by using time in your calculations (@TIME, @TICK, @ONLINE and @IDLE)
- LOCAL variables
- full LOG support
- user definable windows with colours and advanced mouse support
- macros linked to the mouse
- a toolbar to easily manage multiple instances of ELF connected to various servers

Elfred

Line editing

You can use left and right arrows, as well as insert and delete any character you like.

By pressing <ESC> you can clear the text entered.

There are two operating modes for editing: EDITING and DIRECTIONS.

Keys definition:

	EDITING MODE	**DIRECTIONS MODE**
HOME	go to start of line	previous command
END	go to end of line	next command
PAGE-UP		send UP command
PAGE-DOWN		send DOWN command
LEFT	char left	send WEST command
RIGHT	char right	send EAST command
UP	previous command	send NORTH command
DOWN	next command	send SOUTH command
CTRL-LEFT	word left	left char
CTRL-RIGHT	word right	right char
CTRL-HOME	go to start of line	go to start of line
CTRL-END	go to end of line	go to end of line

NOTE: by pressing <CTRL-CANC> some keys are overridden and their meaning changes both in EDITING and DIRECTIONS mode to:

HOME	go to start of line
END	go to end of line
LEFT	left char
RIGHT	right char
UP	previous command
DOWN	next command
CTRL-LEFT	previous word
CTRL-RIGHT	next word

Pressing <CTRL-CANC> again the keys definition reverts to normal.

NOTE: the effect of <CTRL-CANC> automatically ends when the line is sent to the MUD.

See also, Special keys.

Triggers

What is a trigger?

Well... when you are playing MUDs you send commands and receive messages.

When some messages arrive, you do specific actions (send specific commans).

Some messages occur often and the related actions are always the same.

If you receive a message telling you that you're hungry, you issue a command to eat something.

A trigger helps you in this job by comparing a sequence of characters with every line coming from the MUD. If a match is found then the releated commands are sent to the MUD.

Some messages vary slightly (they are not static like 'You are hungry.'). For example a message might look like 'Elfred starts following you.' or 'Velle starts following you.'.

In such case you still want to issue some commands (like 'group Elfred' or 'group Velle') and a trigger ought to be useful. It does!

You may define a trigger containing *variable parts* (also referenced as *parameters*). In the example above, 'Elfred' and 'Velle' are variable parts.

ELF recognises variable parts via the special character '*' (much like DOS' DIR command does). So, if you want to issue some commands when '*anybody* starts following you.' you simply have to define a trigger activated (triggered :-)) by '* starts following you.'. The special character '*' may appear several times in a trigger to manage complex messages with several variable parts ('Elfred hits Velle very hard.' can be recognised by '* hits * very hard.').

If you use variable parts in your triggers (but the same thing may be done with aliases) you'll obviously want to be able to modify the related commands accordingly. That can be achieved via *variable parts extraction*.

There are two ways to achieve *variable parts extraction*:

1 by using the special character '%' followed by some *specifiers*;

2) by using a function (@PARMS or @PARMN depending on the kind of parameter you need to extract: numeric o string).

The two methods are both valid and recognised by ELF, but with some slight, but still important, differences.

First of all, let's see how to use both methods.

The special character '%' can be put in the commands related to a trigger exactly where the variable part ought to be inserted. The first parameter (variable part) is referenced by '%1a', the second by '%2a' and so on.
In the case of a trigger activated by '* starts following you.', the related command might be 'group %1a'.

Let's see how it works!
ELF receives 'Elfred starts following you.'.
Checks it with all defined triggers and matches it with '* starts following you.'.
ELF looks at the commands related to the activated trigger ('group %1a') and finds the special character '%'.
Before issuing such command to the MUD, ELF extracts the specified variable part and builds the command 'group Elfred' that, finally, can be sent to the MUD.

The special character '*' matches any characters sequence (even an empty one).

We spoke about *specifiers*. The '%' character must be followed by some specifiers that instruct ELF on which parameter to extract and how.
Such specifiers are built up in the following way:

- an optional number between '1' and '9' that indicates which parameter you want (when you use multiple '*'). If this number is not specified '1' is used by default;
- an optional '=' character that instructs ELF not to remove trailing and leading spaces from the variable part. If this character is not specified, spaces are removed by default;
- a mandatory character indicating how the variable part must be extracted:
- '%a' that means: ALL the variable part
- '%t' that means: TRANSLATE the variable part
- '%f' that means: the FIRST word of the variable part
- '%l' that means: the LAST word of the variable part

- '%x' that means: the WHOLE line.
  This must be used with great care, because a very long line might be truncated.

So, valid commands might look like:
- '*group %t*'
- '*group %1t*'
- '*group %a*'
- '*group %1=a*'

The alternate method to do variable parts extraction is via the functions @PARMS and @PARMN.

Those functions must be inside an expression. That means that you can't issue a command like '*group @PARMS(1t)*' because this doesn't involve an expression evaluation.

On the other side, when you issue a meta-command, ELF uses expressions evaluation to pick the arguments of the meta-command and so you're free (and encouraged) to use @PARMS and @PARMN instead of '%'.

The argument to those functions is the same as the specifier used after '%' (as listed above).

So valid uses of @ParmS and @ParmN might look like:
- @TempS=@ParmS(t)
- @OUTSTR "Hello "+@Parms(1=a)
- @num=@ParmN(2a)

Remember that @ParmS returns a *string* value and @ParmN returns a *numeric* value.

You can enable/disable ALL triggers at once with the checkbox in the main window named TRIGGERS ON.

NOTE: triggers are checked in the same order you see them during editing. By using this feature, you can build complex behaviours. E.g.: you might use a trigger activated by '*', that means always, to clear some variables to be used by other triggers.

In this way you can have triggers excluding each other, by using variables to control the flow.

INFORMATION: we can specify a minimum time (seconds) between trigger's

activation. This way we can avoid keeping activating a trigger a lot of consecutive times.
If, for instance, we set a trigger to fill a barrel whenever we see a fountain, we might want to set the minimum time between activations to 300 seconds, being, thereafter, able to move around the fountain without keeping on filling the barrel.
In the same way, we might set the trigger to waste MANA (for TICK's recognition) in a way so that it doesn't activate before 10 seconds after the last activation. In this way we can manage network lags.

INFORMATION: it is possible to set a delay between the matching of the trigger and the sending of the command to the MUD. In this way if, for example, we are sleeping, we have the time to wake and stand up.

There are several modifiers that let you fine tune your trigger:

CHECK COMPLETE LINES
REMOVE ORIGINAL LINE
ENABLE RE-CHECKING ON THE SAME LINE
CHECK FOR PROMPT
LAST TRIGGER TO CHECK IF ACTIVATED
TRIGGERS ACTIVATED BY '*'
ORDERED EXECUTION OF TRIGGERS
TRIGGER'S QUANTITY RELATED TO SPEED

CHECK COMPLETE LINES

The matching of the trigger has to take place only when a whole line has been received.

See also, CHECK COMPLETE LINES.

REMOVE ORIGINAL LINE

The matched line must be neither showed nor archived in the SCROLL-BUFFER.

This option is effective only when CHECK COMPLETE LINES is on.

See also, Triggers, REMOVE ORIGINAL LINE.

ENABLE RE-CHECKING ON THE SAME LINE

ELF checks triggers at the end of every line and whenever it is idle (no characters are arriving from the MUD and no key is pressed).

By default it avoids re-checking a trigger on the same line.

If you want to re-check a trigger on a line where it has been already checked, use this option.

This option is very useful for "time-dependant" triggers (those activated by '*'), that don't care what characters have been received.

See also, REPEATED RECOGNITION OF A TRIGGER ON THE SAME LINE.

CHECK FOR PROMPT

If this option is enabled, ELF checks for the presence of the prompt, whatever pattern has been set for matching.

This is useful for fast and easy prompt recognition.

See also, Triggers.

LAST TRIGGER TO CHECK IF ACTIVATED

If a trigger is activated and it has this option enabled, no more triggers will be checked for that line.

USE WITH CARE!!!

See also, LAST TRIGGER.

Macros

MACROs are character sequences assigned to special keys.

If a macro is set as AUTO-EXECUTABLE it will be directly sent to the MUD without interfering with the text in the editing buffer; otherwise it will be appended to it.

MACROs can contain variables that will be expanded at execution time. A macro like 'GET ALL.COIN @DEAD' might be used after every death to get money from the corpse. Of course, we must have assigned the right value to the variable @DEAD when we have found someone's death (by using a trigger).

Special keys

<CTRL-B>	enable/disable buffered input mode
<CTRL-E>	executes @EDITGROUP
<CTRL-I>	to read the configuration of another playing character
<CTRL-J>	view/edit mouse actions
<CTRL-K>	sends to the MUD any string. It is possible to use special chars, by using '#' followed by the ASCII code of the char ('h#97llo#13' sends 'hello' followed by the <ENTER> code)
<CTRL-L>	open/close LOG
<CTRL-N>	resets tick's estimated duration to 75 seconds
<CTRL-O>	view/edit macros
<CTRL-P>	to connect
<CTRL-R>	view/edit translations
<CTRL-T>	view/edit/enable trigger
<CTRL-V>	view/edit variables
<CTRL-Y>	view/edit aliases
<CTRL-CANC>	temporarily turns to EDITING editing mode. To step back to DIRECTIONS editing mode either press <CTRL-CANC> once more or press <ENTER>
<SHIFT-TAB>	turns to HIDDEN input and back
<PAUSE>	FREEZEs receiving chars from the MUD

See also, Line editing.

Translations

MUD uses several names to represent and interpret the different objects and characters. Whenever a description of something we are watching appears, MUD generally uses an extended description.

Yet if we wish to capture that object we often realise that its reduction to a single word may be difficult. What we generally do is to ask its name by SHOUTing to the world and waiting. If the object is unusual we'll be in the same situation in a few days. ELF obviates all this by managing a database of translations from extended to short name.

A different case, more frequent and easier to automate, needing the translation of an extended into a short name, occurs whenever we kill a character such as, for instance, '*The Ghost of Brackenred*'. The moment he dies we'll want to get automatically the money from its body or whatever else. In such cases we'd wish ELF would use the correct name for each character. The translation table allows solving this problem. Using the '%t' parameter when writing the trigger we'll force ELF to translate '*The Ghost of Brackenred*' into the corresponding short name '*bones*'.

Whenever ELF finds a non-recorded extended name, it automatically creates a short name using the extended name's last word. In this case, however, ELF will remember that such a name is unsafe and when exiting the program ELF will check for unsafe names, in which case it will open a special window to validate all the names in the database.

The window will list all the translations the program knows. The unsafe ones will be highlighted in red.

Variables

ELF allows you to use as many variables as you like. Variables can be used to speed up frequently used commands.

EXAMPLE: if the IT variable has the value DEATH-KNIGHT, the command *CAST 'METEOR SWARM' @IT*, will be converted to *CAST 'METEOR SWARM' DEATH-KNIGHT*.

Hereafter I'll show you a better example of using variables:

- let's define a trigger named DEATH so that it recognises a MOB's death ('* *is dead. R.I.P.*');
- let's define the associated command the following way: *@dead="%t" ^ get all.coins @dead ^ save*
- when the trigger will be activated, the command will be expanded and MOB's long name will be translated to the short name (%t). Thereafter the commands will be sent to the MUD;
- the first command ('*@dead="%t"*') is an assignment meta-command; ELF will set DEAD variable to MOB's name; it is a meta-command because it WON'T be sent to the MUD, as it is an internal command directly understood by ELF;
- when ELF will execute '*get all.coins @dead*', the DEAD variable will be substituted by its value, so that, if the ghost of brackenred died, @dead will have been given the value 'bones', and the command will be changed to 'get all.coins bones';
- when the trigger has been correctly matched and executed, we'll be able to get all from the corpse by using a MACRO like '*get all @dead*';

Variables exist throughout ELF's execution and are saved, with their content, to disk when exiting the program.

When you'll use ELF again, ALL the variables will resume the values they had the last time we used the program.

NOTE: variables MUST be manually DEFINED. They are not automatically defined if not existing. I.e.: if DEAD isn't a variable, '*@DEAD="pippo"*' won't be recognised as an assignment meta-command and will be sent AS IS to the MUD.

INFORMATION: you can monitor whatever variable you like. Use the option MONITORIZED. Monitorized variables are shown right above the horizontal toolbar. The size of the panel used for monitorized variables changes dynamically according to the higher Y value used by monitorized variables.

ATTENTION: the most common error is that of not writing the '@' right before the variable name. Beware!

See also, Static variables, Variables used by ELF.

Meta-commands

ELF recognises META-COMMANDS.

When we send a command like: 'get all.coins bones', we are sending a command to the MUD.

Commands starting with '@' are called META-COMMANDS.

META-COMMANDS are not sent to the MUD, but, on the contrary, they are internally interpreted by ELF for special purposes. E.g.: @BEEP sends nothing to the MUD, but plays a short sound.

META-COMMANDS known to ELF are:

@WHILE
@WINSERT
@WPUT
@WSHOW

You can view this list ordered by type

Meta-commands

ELF recognises META-COMMANDS.

When we send a command like: 'get all.coins bones', we are sending a command to the MUD.

Commands starting with '@' are called META-COMMANDS.

META-COMMANDS are not sent to the MUD, but, on the contrary, they are internally interpreted by ELF for special purposes. E.g.: @BEEP sends nothing to the MUD, but plays a short sound.

META-COMMANDS known to ELF are:

assign

Flow control

@CALL
@ENDW
@IF
@QUEUE
@SKIP
@WHILE

Group

@ADDGROUP
@CLEARGROUP
@EDITGROUP
@REMGROUP

User windows

@WADD
@WCLEAR
@WCLOSE
@WHIDE
@WINSERT
@WPUT
@WSHOW

Sounds and output

@BEEP
@OUTMSG
@OUTSTR
@TOCLIP

Files

@FCL
@FWR

Miscellaneous

@ASSIGN
@DELAY
@FWRITE
@LOG
@NEWLINE
@NLOCAL
@REM
@SEND
@SLOCAL

You can view this list ordered by name

@FWRITE

@FWRITE <file>,<text> writes <text> at the end of <file>.

ATTENTION : @FWRITE doesn't automatically insert an end of line. You have to add #13 to do so.

See also, Meta-commands.

@ASSIGN

@ASSIGN(<variable-name>,<expression>) assigns the value of the <expression> to the specified variable. <variable-name> must be of type string. <expression> must be of the same type of the variable assigned to.

@ASSIGN("num",123)
acts like
@num=123

@ASSIGN("TempS","Hello "+"world")
acts like
@TempS="Hello "+"world"

The advantage is that the name of the variable may be changed during execution and thus vary dynamically.

See also, Meta-commands, assign.

@QUEUE

@QUEUE <delay in seconds>,<commands>

Executes the specified commands after the specified number of seconds.

Consider that the commands are read from a string expression, therefore you can store them in a variable or write them directly.

You may encounter some trouble when using multiple commands. Consider that *@CHR(94)* inserts in your string expression the "^" character that is what is needed to separate two lines. If you want to insert a """ character (double-quotes), use *@CHR(34)*.

See also, Meta-commands, @CALL.

@DELAY

@DELAY <delay in milliseconds>

See also, Meta-commands.

@FWR

@FWR <id>,<text> writes *text* in the file identified by *id*.

See also, Meta-commands, FILE BASICS.

@FCL

@FCL <id> closes the files identified by *id*.

See also, Meta-commands, FILE BASICS.

@CONNECT

@CONNECT <name> attempts to connect to the specified MUD. <name> is a string containing the name of one of the MUDs known to ELF in the connection dialog. When the connection is established, the alias OnConnect is called.

See also, Meta-commands, @DISCONNECT.

@DISCONNECT

@DISCONNECT closes the connection to the current MUD. If no MUD is connected, nothing happens. When disconnection has taken place, the alias OnDisconnect is called.

See also, Meta-commands, @CONNECT.

@TOCLIP

@TOCLIP "*text*" puts *text* in the CLIPBOARD.

See also, Meta-commands.

@SEND

Evaluates the subsequent string expression and sends it to the MUD, translating special characters ("*ATZ#13*" is translated to "*ATZ*" followed by char whose ASCII code is 13).

NOTE: after the string, CR is NOT automatically appended. If desired, it must be specified, ending the string by '*#13*'.

See also, Meta-commands.

@OUTSTR

Apprises the subsequent string expression and outputs it to the screen. The string may contain special codes, specified by the symbol '#' and the corresponding ASCII code.

EXAMPLE: @OUTSTR "*#27[41mTest*" outputs to the screen the word '*Test*' in red, since an ESCAPE sequence was specified, setting a red background.
The ESCAPE sequence must begin by the code for the ESCAPE character (=27).

See also, Meta-commands.

@OUTMSG

Apprises the string subsequent expression and outputs it to the screen's bottom line (the copyright one).

See @OUTSTR

See also, Meta-commands.

@ADDGROUP

Apprises the subsequent string expression and adds it to the group, avoiding duplications.

See also, Meta-commands.

@REM

The rest of the line after this meta-command is ignored.
Useful to insert a comment.

See also, Meta-commands.

@REMGROUP

Apprises the subsequent string expression and removes it from the group.

See also, Meta-commands.

@EDITGROUP

Use this to interactively edit the members of your group (same as CTRL-E).

See also, Meta-commands.

@CLEARGROUP

Cancels all present group members.

See also, Meta-commands.

@LOG

Changes the name of LOG FILE and attempts to open it immediately. If an error occurs, ELF will go back to the former name and, if in case, attempts to reopen it. (see CTRL-F).

See also, Meta-commands.

@BEEP

It BEEPS :-)))
This meta-command has a double syntax:

- @BEEP
- @BEEP <soundfile>, where <soundfile> can be .MID or .WAV

See also, Meta-commands.

@SLOCAL

@SLOCAL var1,var2,...,@varN creates @VAR1, @VAR2, up to @VARN as LOCAL STRING variables. Such variables substitute for the existing ones by the same name and may act like any other variable. LOCAL variables exist for the entire execution of the series of linked commands. Should a @CALL be executed or an alias be called, local variables would survive in such sub-programs (where other local variables may be defined, even by the same name).

See also, Meta-commands.

@NLOCAL

Acts like @SLOCAL, but defines NUMERICAL variables.

See also, Meta-commands.

@CALL

@CALL @VariableName executes the commands contained in the referenced variable. Commands may be separated with #13 or with '^'.

This command is very useful to define sequences of commands that can change at run-time.

See also, Meta-commands, Variables used by ELF.

@SKIP

@SKIP n causes the next *n* commands to be skipped (not executed).

See also, Meta-commands.

@IF

Apprises the following numerical expression. If its value equals 0, then the subsequent two commands will be skipped.

EXAMPLE: *@IF @dead="lamia"*
get all.po @dead
@SKIP 1
get all.coi @dead
save
This sequence of commands, that might refer to the trigger activated by someone's death, checks if the dead is a LAMIA.
If so, then all potions are taken from its body, otherwise all coins are taken.
If it is not a LAMIA, the two commands (*get all.po @dead* e *SKIP 1*) following the @IF are skipped. If it is a LAMIA, ALL the commands are executed.
In this example @SKIP 1 has the same effect of an IF-THEN-ELSE structure.
Please note that the SAVE command is executed in both cases.

See also, Meta-commands.

@WHILE

The @WHILE meta-command is paired by @ENDW. It is useful to create a loop.

@WHILE loop **CAN'T BE NESTED**.

EXAMPLE:
```
@SLOCAL x
@x=1
@WHILE @x<=10
@OUTSTR @NTOS(@x)
@x=@x+1
@ENDW
```

NOTE: be careful when writing such loops. If you don't update the variables involved in the test, it will lead to an endless loop.

See also, Meta-commands.

@ENDW

See @WHILE

See also, Meta-commands.

@WCLEAR

@WCLEAR <id> clears user window *id*.

See also, Meta-commands, PROGRAMMING USER WINDOWS.

@WCLOSE

@WCLOSE <id> clears user window *id*.

See also, Meta-commands, PROGRAMMING USER WINDOWS.

@WHIDE

@WHIDE <id> hides user window *id*.

See also, Meta-commands, PROGRAMMING USER WINDOWS.

@WINSERT

@WINSERT <id>,<line #>,<text> writes *text* in user window *id* on the *line #*. The following lines are moved down. The first line number is 0.

See also, Meta-commands, PROGRAMMING USER WINDOWS.

@WPUT

@WPUT <id>,<line #>,<text> writes *text* in user window *id* on the *line #*. The first line number is 0.

See also, Meta-commands, PROGRAMMING USER WINDOWS.

@WADD

@WADD <id>,<text> writes *text* in user window *id* on the line beyond the last one.

See also, Meta-commands, PROGRAMMING USER WINDOWS.

@WSHOW

@WSHOW <id> shows (unhides) user window *id*.

See also, Meta-commands, PROGRAMMING USER WINDOWS.

@NEWLINE

@NEWLINE "text" changes the content of the line on which triggers are being checked.

After executing @NEWLINE "text", subsequent triggers will be checked on the new text.

EXAMPLE: text to be checked: "*This is the original line*"
trigger 1 checks "*This is the original line*"
trigger 2 checks "*This is the original line*"
trigger 2 executes @NEWLINE "*Replacement!*"
trigger 3 checks "*Replacement!*"

NOTE: if your MUD sends to you text and/or commands pasted to the prompt, you can define a trigger that finds the prompt, outputs it to the screen with @OUTSTR and then issues @NEWLINE "<new line>", where <new line> is the original trigger line with the prompt stripped.
Just remember that the last trigger will have to output such a line with @OUTSTR and have Last trigger to check if activated ON to avoid that ELF outputs the original line to the screen.

See also, Meta-commands.

ANSI color codes

ANSI color codes are specified by using an ESCAPE sequence.

In whatever you send to the screen you can add ANSI color codes (even modifying the lines received from the MUD via @NEWLINE), thus highlighting text, names, etc...

An escape sequence starts with the ESC character (ASCII code 27, thus #27) followed by '[', a series of one or more COLOR CODES separated by ';', and a 'm',

'#27[33;44m' sets the background color to BLUE and the foreground to YELLOW.

Recently have been added a lot of new functions that make easier to work with colors. These functions are:

@BGxxx	from @BGBlack to @BGwhite. Set the background color accordingly.
@FGxxx	from @FGBlack to @FGWhite. Set the foreground color accordingly.
@HiColors	selects high intensity colors. This code is almost unuseful, due to the direct color color (what I call IBM COLOR CODES).
@NormalColors	resets background and foreground colors to their default values.
@PushColors	saves colors to a stack for later retrieving. Useful to temporarily change a color, being able to restore the old one later in the line.
@PopColors	retrieves the colors previously saved to the stack.

Using these new functions, the same result as '#27[33;44m' , that sets the background color to BLUE and the foreground to YELLOW, is achieved by @BGBlue+@FGYellow.

ANSI CODES: - the basic colour codes are:

- 0 --> black
- 1 --> red
- 2 --> green
- 3 --> yellow
- 4 --> blue
- 5 --> magenta
- 6 --> cyan

- 7 --> white
- add 30 to get a FOREGROUND colour
- add 40 to get a BACKGROUND colour

IBM COLOURS : - standard IBM colors are 16:

- 0 -->	black	- 1 -->	blue
- 2 -->	green	- 3 -->	cyan
- 4 -->	red	- 5 -->	magenta
- 6 -->	brown	- 7 -->	light grey
- 8 -->	dark grey	- 9 -->	light blue
- 10 -->	light green	- 11 -->	light cyan
- 12 -->	light red	- 13 -->	light magenta
- 14 -->	yellow	- 15 -->	white

- add 50 to get a FOREGROUND colour using these codes
- add 70 to get a BACKGROUND colour using these codes

Assign

A command like '@dead="bob"' assigns the value *bob* to the variable DEAD.

ATTENTION: if such a variable doesn't exist, the META-COMMAND won't be recognised and will be sent to the MUD as is.

NOTE: if the assign command is recognised, nothing will be sent to the MUD.

Assigning value to variables, you can take advantage of ELF's capability of evaluating expressions, both numerical and string type.

EXAMPLE: (complex, but nice)

- let's suppose that both DUMMY and TEMPS are existing variables declared with type STRING;
- let's create a trigger:
  - activated by:
    **says: Let me pass, you beggars!*
  - and that executes:
    @dummy="%t"
    cast 'power word kill' @dummy
    @TempS="Beggar?!? To whom?!? You silly "+@dummy+"! :-)"
    gos @TempS

- you guess what follows :-)))

IMPORTANT: you can use a special string assign command.
Instead of '=' you can use '$'.
By using '$' you tell ELF to assign WHATEVER follows to the variable.
In this way you can assign special characters received from the mud, such as "" and so on.

See also, @ASSIGN.

Aliases

An ALIAS is a sequence of commands with optional parameters interpreted internally by ELF.

NOTE : aliases are checked in the same order they appear in the configuration page.
By using this feature you can create complex behaviours.
You can, for example, define an alias activated by the pattern '*help * barely,**' and another one activated by the pattern '*help *,**'. In this way you can distinguish between 'HITS' and 'BARELY HITS' very easily and calling the same alias.

It is important to know that, unlike triggers, when an alias is executed, no more aliases are checked.

Using an ALIAS you can create a new command with parameters.
An ALIAS is a command recognised by ELF because it starts with '<'.
Basic to the ALIAS is the PATTERN which identifies the ALIAS.
The PATTERN defines both ALIAS's name and its grammatical structure.

EXAMPLE: I want to get something from my leather pouch (that I hold on my back) and keep it in my inventory in place of another thing. I'll have to execute some commands like these:

rem pou^get staff pou^put sword pou^wea pou

If now I would like to exchange two other objects, I'd have to repeat almost all these commands with minor changes.
Let's define an ALIAS:

- PATTERN: *swap * **
  This is a PATTERN defining ALIAS's NAME (SWAP) and its two parameters (the two '*').
  If ELF recognises an ALIAS's PATTERN, then it executes the related command. The command can have parameters much like triggers). These parameters will be substituted before executing.
- COMMANDS: *rem pou*

  take %1a
  put %2a pou

wea pou'
The '%' identifies a parameter. The following number indicates which parameter to use (every '*' means a parameter). The ending 'a' means that the parameter must be taken without further changes. You can even use parameters like '%1t', where 't' forces parameter's translation (*The Ghost of Brackenred* becomes *bones*)

If you write '<swap sword staff', you'll get exactly what we liked to get.

NOTE: ALIAS's definition is built up using a PATTERN, and not in an easier way, to avoid limiting ALIAS's power.
For example, I could have forced the parameters to be separated ALWAYS by blanks, but, then, what about a parameter like '*hello world*'?
In the former example we could have defined a pattern 'swap *,*', allowing the usage of parameters with blanks inside.

WARNING: if you define an ALIAS with the pattern 'test *' and then you try to execute '<test ' ('test' followed by two blanks), the alias WON'T be activated because ELF always removes trailing and leading spaces from commands. So your '<test ' becomes '<test' and there is no pattern matching...
To avoid this, use a pattern like this: 'test "*"'. In this way you'll call the alias with '<test " "' and everything will be right :-)

See also, ALIASES UNVEILED, Aliases used by ELF, Extended pattern.

Extended pattern

With this option you can force ELF to expand the variables contained in the pattern of an alias each time the pattern is checked.

In this way you can parametrise aliasesaliases in a very powerful way.

Suppose that you want to trap *<OnWinClick*.

You might want to write an alias that traps it only for a specific window (say the one whose *id* is contained in the *@EqWin* variable).

Without using this flag you ought to write such an alias in this way:

Pattern: *<OnWinClick *,*,**
Commands: *@IF @EqWin=%1a*
@OUTSTR "Right window!"

Problems arise if you would like to handle *<OnWinClick* for one more windows... You would have to add new tests in the commands of the alias above.

By using the *extended pattern* option you could solve all of these problems. You simply had to define the former alias in this way:

Pattern: *<OnWinClick @EqWin,*,**
Commands: *@OUTSTR "Right window!"*

A similar alias can be defined for every other window you want, without interfering each other!

ATTENTION: this flag is very powerful but ***slows ELF down*** because ELF has to manipulate the pattern every time it needs to check it. To reduce such overload you can put such aliases near the bottom of the aliases. In this way it is likely that other aliases are recognised earlier in the list, thus preventing ELF from further pattern checking.

Expression evaluation

Whenever ELF expects numerical or string values it appraises the expressions through operators and FUNCTIONS.

String operators
Numeric operators

Functions

String operators

+	-->	links 2 strings
>	-->	returns 1 if the first string is greater than the second; 0 otherwise
<	-->	returns 1 if the first string is lesser than the second; 0 otherwise
=	-->	returns 1 if the first string equals the second; 0 otherwise
<>	-->	returns 1 if the first string differs from the second; 0 otherwise
<=	-->	returns 1 if the first string is lesser than or equal to the second; 0 otherwise
>=	-->	returns 1 if the first string is greater than or equal to the second; 0 otherwise

See also, Expression evaluation.

Numeric operators

+	-->	adds 2 numbers
-	-->	subtracts 2 numbers
*	-->	multiplies 2 numbers
/	-->	divides 2 numbers (integer division)
#	-->	executes MOD operator (12#5=2)
|	-->	ORs two numbers
&	-->	ANDs two numbers
>	-->	returns 1 if the first number is greater than the second; 0 otherwise
<	-->	returns 1 if the first number is lesser than the second; 0 otherwise
=	-->	returns 1 if the first number equals the second; 0 otherwise
<>	-->	returns 1 if the first number differs from the second; 0 otherwise
<=	-->	returns 1 if the first number is lesser than or equal to the second; 0 otherwise
>=	-->	returns 1 if the first number is greater than or equal to the second; 0 otherwise

See also, Expression evaluation.

Functions

@BGxxx	@BGxxx functions let you easily define BACKGROUND colors in strings that will be output to the screen. xxx can be one of the 16 standard IBM colors. EXAMPLE: to set the background color of some text to LIGHT RED all you'll have to write will be: @BGLightRed+"text". If you want to output "Hello Alfredo" on a line where "Alfredo" has to be written with a MAGENTA background, the command will look like: @OUTSTR "Hello "+@BGMAGENTA+"Alfredo"
@CHR	returns the char with the corresponding ASCII code (usage: @CHR(number))
@CRY	@CRY(<text>,<key>) encrypts <text> using the specified key. *Text* and *key* are both strings. Returns a string. This function encrypts text in the range chr(32)..chr(127) (printable characters) returning text in the same printable range.
@DCRY	@DCRY(<text>,<key>) decrypts <text> using the specified key. *Text* and *key* are both strings. Returns a string.
@FGxxx	@FGxxx functions let you easily define FOREGROUND colors in strings that will be output to the screen. xxx can be one of the 16 standard IBM colors. EXAMPLE: to set the foreground color of some text to LIGHT CYAN all you'll have to write will be: @FGLightCyan+"text". If you want to output "Hello Alfredo" on a line where "Alfredo" has to be written with a red foreground, the command will look like: @OUTSTR "Hello "+@FGRED+"Alfredo"
@FOPW	@FOPW(<file-name>) opens <file-name> for writing. Returns a numeric *id* (0 if an error occurs). If the file already exists, it is deleted before rewriting it.
@FOPR	@FOPR(<file-name>) opens <file-name> for reading. Returns a numeric *id* (0 if an error occurs). The file must exist.
@FOPA	@FOPA(<file-name>) opens <file-name> for appending. Returns a numeric *id* (0 if an error occurs).
@FNRD	@FNRD(<id>) reads a number from the file identified by *id*.
@FSRD	@FSRD(<id>) reads a string from the file identified by *id*.
@FEOF	@FEOF(<id>) returns 1 if the file identified by *id* has reached its end, 0 otherwise.
@GETDAY	returns the day of the month

@GETMONTH	returns the current month
@GETYEAR	returns the current year
@HICOLORS	this function exists only for full compatibility with past versions of ELF. Subsequent ANSI colors will be shown in high intensity format. The presence of @BGxxx and @FGxxx functions obsolete this function.
@IDLE	returns the number of seconds since the last key was pressed
@INGROUP	checks whether the specified string is a member of your group. Returns 0 if it isn't, its index in the group if it is (usage: @INGROUP(string))
@LASTMUD	returns the name of the last mud you connected to since when ELF was opened
@LEN	returns string length (usage: @LEN(string))
@MYHP	@MYHP returns the value of your HPs that ELF has extracted from the PROMPT (you must correctly set the relevant parameters in the general configuration page).
@MYMANA	@MYMANA returns the value of your MANA that ELF has extracted from the PROMPT (you must correctly set the relevant parameters in the general configuration page).
@MYMOV	@MYMOV returns the value of your MOVement that ELF has extracted from the PROMPT (you must correctly set the relevant parameters in the general configuration page).
@NORMALCOLORS	resets the colors in some output text to default values. I.e.: @OUTSTR @FGRed+"Hello "+@NormalColors+"Alfredo" will display "Hello " with a default background and a red foreground and "Alfredo" with default background and foreground.
@NTOS	converts a number into a string (usage: @NTOS(number))
@ONLINE	returns 1 if you are connected to a MUD, 0 otherwise
@PARMS	@PARMS(<parameter>) returns the value of a parameter (much like '%') as a string. @ParmS(2=a) equals to '%2=a' but is faster and completely avoids maximum string length overflow (255 chars). Inside the parenthesys you'll have to put the same characters that you would have put after the '%' sign to extract a parameter in a trigger or alias. See trigger for more informations on parameter extraction. This function has several advantages over the older parameter extraction method (via the '%' sign). It is faster, has type checking (@ParmS returns a string value) and IS NOT EXPANDED inside

	the commands but during expression evaluation. That means that you no longer will exceed the maximum length of 255 characters when referencing a very long parameter. As a side effect, you no longer will be forced to assign such parameters to temporary variables to avoid exceeding, thus gaining speed and clarity.
@PARMN	@PARMN(<parameter>) returns the value of a parameter (much like '%') as a number. @ParmN(2=a) equals to '%2=a' but is faster and completely avoids maximum string length overflow (255 chars). See @PARMS for more informations.
@POPCOLORS	Restores the colors previously saved via @PUSHCOLORS.
@POPUPS	@POPUPS("option-1",..,"option-n") displays a popup menu and returns the name of the selected item. If no item is selected, returns an empty string.The options must be of type string and may result from expression evaluation. The popup menu is shown at mouse position.
@POPUPN	@POPUPN("option-1",..,"option-n") displays a popup menu and returns the index (the first item has an index value of 1) of the selected item. If no item is selected, returns 0. The options must be of type string and may result from expression evaluation. The popup menu is shown at mouse position.
@POS	@POS(<text>,<source>,<start>) returns the position of <text> in the <source> string starting the search from the <start>th character. Returns 0 if no match is found.
@PUSHCOLORS	Saves the colors used at a certain point in an output string for later recall via @POPCOLORS. Useful to temporary set some colors inside other unknown colors. EXAMPLE: if you want to highlight a word in some text without interfering with other colors: @OUTSTR @FGGreen+"Hello "+@PushColors+@FGRed+"my"+@PopColors+" dear." will output "Hello " and " dear." with a green foreground color and "my" with a green foreground color.
@RND	@RND(<max>) returns a random value between 0 and max-1. Max must be in range 0..65000.
@STON	converts a string into a number (usage: @STON(string))
@TICK	returns the estimated TICK duration in seconds
@TICKS	returns the number of TICKs recognised by ELF
@TIME	returns the hour in terms of seconds after midnight

@TOTICK	returns the number of seconds to the end of the TICK
@WGET	@WGET(<id>,<line #>) returns the content of the line of the user window.
@WLINES	@WLINES(<id>) returns the number of lines in the user windows.
@WMAXID	@WMAXID returns the highest *id* allocated at that moment by ELF for user windows. This is useful to scan all of them together with @WSTAT.
@WNUM	@WNUM(<title>) returns the *id* of the user window with that title. Returns -1 if such a window doesn't exist.
@WOPEN	@WOPEN(<title>) opens a user window with the specified title and returns an *id*. ELF remembers position and dimension (referenced by title) of the user windows.
@WOPEQ	@WOPEQ(<title>) opens a user window with the specified title and returns an *id*. If a user window with the same title already exists then returns that *id*. ELF remembers position and dimension (referenced by title) of the user windows.
@WSTAT	@WSTAT(<id>) returns a value depending on the visual status of the user window referenced by *id*: 0 - doesn't exist 1 - iconicized 2 - hidden 3 - normal 4 - zoomed
@WTITLE	@WTITLE(<id>) returns the title of the user window referenced by *id*. Returns " if such a window doesn't exist.

ATTENTION: expressions are appraised irrespective of some operators's priority over others. Use parentheses to force priority as needed.

You can view this list ordered by group.

See also, Expression evaluation.

Functions

User windows:

@WGET	@WGET(<id>,<line #>) returns the content of the line of the user window.
@WLINES	@WLINES(<id>) returns the number of lines in the user windows.
@WMAXID	@WMAXID returns the highest *id* allocated at that moment by ELF for user windows. This is useful to scan all of them together with @WSTAT.
@WNUM	@WNUM(<title>) returns the *id* of the user window with that title. Returns -1 if such a window doesn't exist.
@WOPEN	@WOPEN(<title>) opens a user window with the specified title and returns an *id*. ELF remembers position and dimension (referenced by title) of the user windows.
@WOPEQ	@WOPEQ(<title>) opens a user window with the specified title and returns an *id*. If a user window with the same title already exists then returns that *id*. ELF remembers position and dimension (referenced by title) of the user windows.
@WSTAT	@WSTAT(<id>) returns a value depending on the visual status of the user window referenced by *id*: 0 - doesn't exist 1 - iconicized 2 - hidden 3 - normal 4 - zoomed
@WTITLE	@WTITLE(<id>) returns the title of the user window referenced by *id*. Returns " if such a window doesn't exist.

Parameters handling:

@PARMS	@PARMS(<parameter>) returns the value of a parameter (much like '%') as a string. @ParmS(2=a) equals to '%2=a' but is faster and completely avoids maximum string length overflow (255 chars). Inside the parenthesys you'll have to put the same characters that you would have put after the '%' sign to extract a parameter in a trigger or alias. See trigger for more informations on parameter extraction. This function has several advantages over the older parameter

extraction method (via the '%' sign). It is faster, has type checking (@ParmS returns a string value) and IS NOT EXPANDED inside the commands but during expression evaluation. That means that you no longer will exceed the maximum length of 255 characters when referencing a very long parameter. As a side effect, you no longer will be forced to assign such parameters to temporary variables to avoid exceeding, thus gaining speed and clarity.

@PARMN	@PARMN(<parameter>) returns the value of a parameter (much like '%') as a number. @ParmN(2=a) equals to '%2=a' but is faster and completely avoids maximum string length overflow (255 chars). See @PARMS for more informations.

File handling:

@FOPW	@FOPW(<file-name>) opens <file-name> for writing. Returns a numeric *id* (0 if an error occurs). If the file already exists, it is deleted before rewriting it.
@FOPR	@FOPR(<file-name>) opens <file-name> for reading. Returns a numeric *id* (0 if an error occurs). The file must exist.
@FOPA	@FOPA(<file-name>) opens <file-name> for appending. Returns a numeric *id* (0 if an error occurs).
@FNRD	@FNRD(<id>) reads a number from the file identified by *id*.
@FSRD	@FSRD(<id>) reads a string from the file identified by *id*.
@FEOF	@FEOF(<id>) returns 1 if the file identified by *id* has reached its end, 0 otherwise.

Date and time:

@GETDAY	returns the day of the month
@GETMONTH	returns the current month
@GETYEAR	returns the current year
@TIME	returns the hour in terms of seconds after midnight

System:

@IDLE	returns the number of seconds since the last key was pressed
@ONLINE	returns 1 if you are connected to a MUD, 0 otherwise
@TICK	returns the estimated TICK duration in seconds
@TICKS	returns the number of TICKs recognised by ELF

@TOTICK	returns the number of seconds to the end of the TICK

Strings and conversions:

@CHR	returns the char with the corresponding ASCII code (usage: @CHR(number))
@LEN	returns string length (usage: @LEN(string))
@NTOS	converts a number into a string (usage: @NTOS(number))
@POS	@POS(<text>,<source>,<start>) returns the position of <text> in the <source> string starting the search from the <start>th character. Returns 0 if no match is found.
@STON	converts a string into a number (usage: @STON(string))

Color handling:

@BGxxx	@BGxxx functions let you easily define BACKGROUND colors in strings that will be output to the screen. xxx can be one of the 16 standard IBM colors. EXAMPLE: to set the background color of some text to LIGHT RED all you'll have to write will be: @BGLightRed+"text". If you want to output "Hello Alfredo" on a line where "Alfredo" has to be written with a MAGENTA background, the command will look like: @OUTSTR "Hello "+@BGMAGENTA+"Alfredo"
@FGxxx	@FGxxx functions let you easily define FOREGROUND colors in strings that will be output to the screen. xxx can be one of the 16 standard IBM colors. EXAMPLE: to set the foreground color of some text to LIGHT CYAN all you'll have to write will be: @FGLightCyan+"text". If you want to output "Hello Alfredo" on a line where "Alfredo" has to be written with a red foreground, the command will look like: @OUTSTR "Hello "+@FGRED+"Alfredo"
@HICOLORS	this function exists only for full compatibility with past versions of ELF. Subsequent ANSI colors will be shown in high intensity format. The presence of @BGxxx and @FGxxx functions obsolete this function.
@NORMALCOLORS	resets the colors in some output text to default values.

	I.e.: @OUTSTR @FGRed+"Hello "+@NormalColors+"Alfredo" will display "Hello " with a default background and a red foreground and "Alfredo" with default background and foreground.
@POPCOLORS	Restores the colors previously saved via @PUSHCOLORS.
@PUSHCOLORS	Saves the colors used at a certain point in an output string for later recall via @POPCOLORS. Useful to temporary set some colors inside other unknown colors. EXAMPLE: if you want to highlight a word in some text without interfering with other colors: @OUTSTR @FGGreen+"Hello "+@PushColors+@FGRed+"my"+@PopColors+" dear." will output "Hello " and " dear." with a green foreground color and "my" with a green foreground color.

Miscellaneous:

@CRY	@CRY(<text>,<key>) encrypts <text> using the specified key. *Text* and *key* are both strings. Returns a string. This function encrypts text in the range chr(32)..chr(127) (printable characters) returning text in the same printable range.
@DCRY	@DCRY(<text>,<key>) decrypts <text> using the specified key. *Text* and *key* are both strings. Returns a string.
@INGROUP	checks whether the specified string is a member of your group. Returns 0 if it isn't, its index in the group if it is (usage: @INGROUP(string))
@LASTMUD	returns the name of the last mud you connected to since when ELF was opened
@MYHP	@MYHP returns the value of your HPs that ELF has extracted from the PROMPT (you must correctly set the relevant parameters in the general configuration page).
@MYMANA	@MYMANA returns the value of your MANA that ELF has extracted from the PROMPT (you must correctly set the relevant parameters in the general configuration page).
@MYMOV	@MYMOV returns the value of your MOVement that ELF has extracted from the PROMPT (you must correctly set the relevant parameters in the general configuration page).
@POPUPS	@POPUPS("option-1",..,"option-n") displays a popup menu and returns the name of the selected item. If no item is selected, returns

	an empty string.The options must be of type string and may result from expression evaluation. The popup menu is shown at mouse position.
@POPUPN	@POPUPN("option-1",..,"option-n") displays a popup menu and returns the index (the first item has an index value of 1) of the selected item. If no item is selected, returns 0. The options must be of type string and may result from expression evaluation. The popup menu is shown at mouse position.
@RND	@RND(<max>) returns a random value between 0 and max-1. Max must be in range 0..65000.

ATTENTION: expressions are appraised irrespective of some operators's priority over others. Use parentheses to force priority as needed.

You can view this list ordered by name

See also, Expression evaluation.

Programming elf

ELF programming is one of the most fascinating, but its most difficult aspects.

By programming we intend the ability to alter the sequential flow of commands.

EXAMPLE: by means of a trigger one can identify the death of a MOB. At this point one can associate the trigger to a series of consequent commands. Generally one will get coins, all the objects, and/or will SAVE.
Having played a not very strong magic user, I could not afford to get everything; thus I chose to get only coins.
Ok thus far. At a certain point, yet, I noticed that LAMIAs carried potions to cure myself. Very useful! But how about it? Having 107 HP's at level 50, Elfred could not waste time to get BOTH *coins* AND *potions* from lamias, since each time he risked being heavily hit. This is the time when programming takes over.
I adopted a command, triggered by the death of a MOB, programmed as follows:

- *@dead="%t"* <-- to assign MOB's name by executing translation
- *@IF@dead="lamia"* <-- to check whether the cadaver was a lamia
- *get all.po lamia* <-- if so, get the potions
- *@SKIP 1* <-- skip the next command to avoid wasting time to get coins (lamias are poor :-))
- *get all.coin @dead* <-- if the cadaver was not a lamia, get coins by this command.
  This command is the third one AFTER @IF and, thus, is the one that is executed when @IF is not verified

Programming allows automating fully many behaviours and exploiting fully the traits/needs of one's character.

For Elfred any spared command was an additional POWER WORD KILL and

a lot of spared HP's.

Exploitation of programming for such cases is still quite easy.
Tackling other problems is more difficult.

EXAMPLE: Pindus (a sweet and powerful cleric) had to HOLD the CAVE FISHER SKULL (anti-drain). You can't hold that skull if you still have something in your hands. So, you first have to remove your weapon, than hold the skull and, at last, re-wield the weapon.
Since Pindus uses either a SWORD or a KRISS, the problem was recognising which weapon he was using.
To solve this problem there are several ways. I'll show you one.
It is very important to create a STRING variable named, for example, *@weapon*. We'll choose as its initial value "*nothing*".
At this point, we'll define a MACRO that will be used to activate the swap operation:

- *@weapon="test"* <-- this is necessary to recognise a simple weapon change from our swap
- *rem sword* <-- let's try to remove the sword...
- *rem kris* <-- ...and the kris too

At this point, the MUD will receive both commands and within some (unknown) time will send the appropriate answers.
Everything will happen in an asynchronous way. I.e.: after REM SWORD we will not receive IMMEDIATELY the answer from the MUD showing what happened (=whether we were wielding it or not).
This is the reason why I use to say that ELF must be programmed by EVENTS.
Now we need 2 triggers to trap both cases:
- trigger for the kris:
  - triggered by: "**You stop wielding the ebony kris**"
  - commands:
    - *@IF@weapon="test"* <-- this check is here to avoid swapping even when we are just stopping wielding the kris
    - *@weapon="kris"* <-- let's store the removed weapon
    - *@CALL @skulla* <-- sub-routine that actually swaps things
    - *@weapon="nothing"* <-- swapping has been recognised. So we can initialise again the variable.

Please note that this is the only executed command when *@weapon* equals "*test*"

- trigger for the sword:
  - triggered by: "**You stop wielding a sword**"
  - commands:
    - *@IF @weapon="test"* <-- same as for the kris
    - *@weapon="sword"* <-- let's store the removed weapon
    - *@CALL @skulla* <-- actually do the swap
    - *@weapon="nothing"* <-- same as for the kris

Now we have to define the variable @skulla, that will contain the sequence of commands needed to hold the skull, that we suppose to be in our leather pouch.
Please note that at this point the variable @weapon holds the name of the weapon we were wielding.
The commands contained in the variable @skulla will be the following:

- *rem pou* <-- let's get the pouch
- *take cave pou* <-- let's take the skull out of the pouch
- *hold cave* <-- let's hold it
- *wield @weapon* <-- let's wield the weapon we were using

NOTE: this procedure works fine only if we had only one hand full, otherwise we wouldn't be able to wield the weapon again.

See also, Advanced programming.

Advanced programming

There are some concepts that must be understood to take full advantage of ELF's features.

See also, Programming ELF.

TRIGGERS ACTIVATED BY ''*

A trigger activated by '*' will be ALWAYS activated. Using some kind of timers we can execute timed operations. It is possible, for example, to SAVE every 5 minutes. Using this feature together with the @IDLE function allows us to create powerful behaviours. We might, for example, automatically SAVE when no key is pressed for more than 15 seconds.

See also, Advanced programming.

ORDERED EXECUTION OF TRIGGERS

Triggers are checked in the same order in which they appear in the editing window.

Any given trigger in the list, set to be activated by '*' (i.e., always), may be used to initialise some variables needed to control the flow of subsequent trigger execution.

EXAMPLE: those triggers that manage AUTO-ASSIST recognise the name of the attacker exploiting a very potent characteristic of aliases.
Thus control of 'HITS', 'BARELY HITS', and so forth, is transferred to aliases. This could be obtained using TRIGGERS, defining one to verify 'BARELY HITS' before another that verifies 'HITS'. The problem arises from the fact that 'BARELY HITS' would be verified by both triggers.
If a potentially ambiguous trigger is verified, one should store this in a variable affecting the execution of subsequent related triggers.

See also, Advanced programming.

REPEATED RECOGNITION OF A TRIGGER ON THE SAME LINE

ELF avoids further verification of a trigger already verified on the same line.

In order to recognise the occurrence of '*Livio*' within the line received by MUD one could simply define a trigger activated by '**Livio**' causing it, for instance, to emit a @BEEP. Should the line look like '*Livio says....*', ELF might verify the occurrence of triggers whenever it received '*Livio* ', '*Livio s*', '*Livio sa*' and so on. Let us mention that verification will SURELY occur at least at the end of the line, while it is LIKELY that it may also occur *within* it. In order to avoid continued or multiple @BEEPs or endless eating or drinking, ELF avoids multiple verification of the same trigger on the same line. If in case one may override this behaviour for any given TRIGGER.

EXAMPLE: a trigger activated by '*' (thus managing a time behaviour) is the best example of a trigger to be repeatedly verified on the same line; otherwise it would be verified only when a new line arrives and not over and over (thus depending on the line flow arriving from the MUD).

See also, Advanced programming.

CHECK COMPLETE LINES

ELF, if no key is pressed and if there is a pause in the flow of the chars arriving from the mud, checks triggers even on partially received lines.

This is useful, for example, for logins. Some times it might be necessary to check triggers only on complete lines.

See also, Advanced programming.

REMOVE ORIGINAL LINE

One might want to change the appearance (even colours) of any incoming message. Simply use a trigger that identifies the line and instructs ELF not to SHOW that line. The meta-command @OUTSTR can then be used to remap the output.

EXAMPLE: immortals speak with mind and a message like this appears: '*::Elfred:: hello boys*'. With a trigger activated by '*::*::**' on a COMPLETE line (another useful flag) one can tell ELF to REMOVE the received line and a nice message can be output, with colours and a text like '*Elfred thinks...*'.

For a successful use of such a trigger, CHECK COMPLETE LINES must be ON, otherwise ELF might check triggers on partially received lines, and messages like '*Elfred thinks 'he"*, instead of '*Elfred thinks 'hello boys"* might arise.

See also, Advanced programming.

@AUTOEXEC

At startup ELF automatically executes *@CALL @AUTOEXEC*. This is a useful feature that might be used, for example, to initialise variables.

See also, Advanced programming.

@AUTOPC

Whenever a new configuration is loaded, with CTRL-I or by clicking on the name of the playing character, executes *@CALL @AUTOPC* after having loaded the new configuration. If @AUTOPC doesn't exist, nothing will happen.

See also, Advanced programming.

WHAT TO DO IF A SERIES OF COMMANDS IS TOO LONG

If you need a very long sequence of commands, you can break it up in smaller parts by using @CALL or, I prefer them, aliases.

By using aliases you avoid the need of defining a new variable containing the commands you need. Furthermore, aliases let you use parameters without the need of passing them with other variables.

See also, Advanced programming.

LOCAL VARIABLES

You often need routines that create intermediate results.
By using local variables you can create them on the fly without interfering with variables with the same name declared externally.

See also, Advanced programming.

ALIASES UNVEILED

Let's define an aliasaliases with the following pattern 'test * *' and the following only command '*@%1a=%2a*'. If we call this alias with '*<test autoassist 1*', we'll force ELF to execute '*@autoassist=1*'. In the same way, with '*<test tempS "duffy"*' we'll force ELF to execute '*@temps="pippo"*'.

This feature can be very useful to specify which variable to use inside an alias or where to put the result.

Combining this feature with local variables lets you define procedures much like PASCAL and other high level languages.

EXAMPLE - PASCAL definition:

PROCEDURE DOUBLE(X:INTEGER;VAR D:INTEGER)

- ALIASaliases definition:
  - pattern:
    - *double * **
  - commands:
    - *@NLOCAL num*
    - *@num=%1a*2*
    - *@%2a=@num*
  - ALIASaliases uses:
    - *<double 27 result*
    - *<double @num result* <-- **NOTE:** *@num* is **not altered** by this alias
    - *<double @num num* <-- **NOTE:** since *@num* is a local variable, the last assignment changes the local instance of *@num* and not the global one.

ATTENTION: That much flexibility has, obviously, its counterpart. If you use such definitions, you'll have to pay a lot of attention issuing commands. I.e.: if we would have issued '*<double 27 5*' instead of '*<double 27 result*', ELF would have **sent to the MUD** '*@5=54*', since it was able to evaluate 27*2 (in the local *@num* variable, later correctly substituted in '*@%2a=@num*'), but it couldn't find the '*@5*' variable!

NOTE: to solve the problem showed by '*<double @num num*', you can define the

previous alias in this way:

- COMMANDS:
  - *@IF %2a="num"*
  - *@SKIP 5*
  - *@SKIP 0*
  - *@NLOCAL num*
  - *@num=%1a*2*
  - *@%2a=@num*
  - *@SKIP 99*
  - *@NLOCAL num1*
  - *@num1=%1a*2*
  - *@%2a=@num1*

See also, Advanced programming.

LAST TRIGGER

ELF is able to highlight the lines containing '*ti*' and '*you*' automatically.

It would be much prettier to highlight them by using triggers so to gain more control over it.

EXAMPLE: I decide to change the color of a message (by removing the original line and showing a new one with @OUTSTR) with a trigger. If the message would contain '*you*', ELF would highlight it by itself making unuseful my previous color selection...

I can disable ELF's automatic highlighting and handle '*you*' messages by myself by using a trigger and putting it near the last trigger in the trigger list.

Now, if any trigger **before** the highlighter one decides to change the appearance of a line, the only thing I will need is to set the last trigger flag **on**. In this way the execution of triggers is stopped and the highlighter one is not checked.

See also, LAST TRIGGER TO CHECK IF ACTIVATED, Advanced programming.

@SKIP AND @IF

@IF is mostly used together with @SKIP to skip a set of commands.
The @IF is executed when the following expression evaluates to 1.
To make the program smaller and somewhat faster, you can take advantage of this feature:

@IF @ONLINE=0
@SKIP 0 <-- equals to --> *@SKIP (@ONLINE=0)*5*
@SKIP 5

If @ONLINE equals to 0, the result of *@ONLINE=0* will be 1, that, multiplied by 5, will lead to @SKIP 5. If @ONLINE is not 0, then *@ONLINE=0* will result in 0, that, multiplied by 5, will lead to *@SKIP 0*.

See also, Advanced programming.

TRIGGER'S QUANTITY RELATED TO SPEED

ELF is not slowed down that much by the presence of a lot of triggers to verify. What slows it down is executing the commands linked to verified triggers.

That means that I can have a lot of triggers seldom triggered without interfering on ELF's speed. In this class of triggers are those used to set *username* and *password*.

The triggers that mostly use ELF's resources are those activated by '*'.

ELF's slowdown can, obviously, be seen on slow computers and fast links.

See also, Advanced programming.

MOUSE AND SCROLL-BUFFER

It seems that you can define only two different functions linked to the mouse pressed over the SCROLL-BUFFER, but this is not completely true!

You simply have to parametrize them...

Let's see how.

We want to parametrize the right mouse button.

Let's instruct ELF to assign the word under the cursor (over the scroll-buffer) to the variable *@TempS*.

Now, let's define, on the vertical or the horizontal toolbar, some buttons that assign different values to the *@RAction* variable.

In the variable *@RAction* we'll put what we want the right mouse button to do.

Now we simply have to instruct ELF to auto-execute "*@CALLmc_call @RAction*" when the right mouse button is pressed over the scroll-buffer.

EXAMPLE - definition of the right mouse button:
- variable to be assigned to:
  TempS
- commands to be auto-executed:
  - *@IF @TempS=""*
  - *@TempS="Elfred"*
  - *@SKIP 0*
  - *@CALL @RAction*

- definition of one command activated by the mouse:
  - name:
    JUNK
  - commands to be auto-executed:
    - *@mouse="JUNK"*
    - *@RAction="junk @TempS"+@CHR(94)*

HOW IT WORKS:
- if the right button definition, the @IF is used so that if we press it over a blank space, a special name is selected (here is is Elfred); this is useful for such commands like "*cast 'ARMOR' @TempS*", because it allows to cast it on ourself too
- the *@mouse* variables is useful if you monitorise it, so that you can see what the mouse button will do
- @CHR(94) is the character that ELF uses to separate commands. It can't be specified directly with #94 because it

would be translated to <ENTER> at the wrong moment...

See also, Advanced programming.

STATIC VARIABLES

Let's suppose to have defined a trigger that calculates how long we've been on-line.

Such a trigger will store the starting hour in a variable and will calculate the connection time when the connection will be lost.

If we change PC's configuration, the value of such a variable will be lost, making our efforts unuseful.

If we define such a variable as a static one, its value will survive through the configuration change.

Obviously the same variable should have to be used by the new configuration :-)

See also, Advanced programming.

HOW TO NEST @WHILE-@WEND LOOPS

ELF doesn't allow you to nest @WHILE-@WEND loops in the same block of commands. You can avoid this by using an alias.

EXAMPLE - main block:

- *@NLOCAL x*
- *@x=1*
- *@WHILE @x<3*
- *<ExtAlias @x*
- *@x=@x+1*
- *@ENDW*

- alias (ExtAlias *):
  - *@NLOCAL c*
  - *@c=0*
  - *@WHILE @c<%1a*
  - *@OUTSTR "%1a."+@NTOS(@c+1)*
  - *@c=@c+1*
  - *@ENDW*
- what happens:
  - the first block of commands counts (*@x*) from 1 to 2 (2<3) and calls the alias with *@x* as a parameter
  - the alias will be called with "<ExtAlias 1" e "<ExtAlias 2" and will output "*1.1*", "*2.1*" e "*2.2*".

See also, Advanced programming.

PROGRAMMING USER WINDOWS

With user windows you can do a lot of complex things. You can:

- show the result of a command in a non-volatile environment, remapping it and filtering unwanted data (i.e.: the result of the STAT command on DikuMUDs is quite long and indecipherable; you can store what you want with the colors you like in a user window);
- automatically issue some commands and completely redirect the output to a user window (i.e.: you might issue the WHO command every 5 minutes (handling it with a timed trigger) and store the result in a user window for later reference);
- store your equipment and automatically give it to the super-guy for repairs with a simple click on the item you like (when the left mouse button is pressed over a user window, the alias '*<OnWinClick WinId,xpos,ypos*' is issued if defined);
- use a hidden user window as a vector. ELF has the ability to convert strings to numbers, so you can define both string and numerical vectors. If you can define a vector you can handle even arrays! With a couple of aliases (and @WPUT and @WGET) it would be almost easy;
- define a list of players with different flags. I.e.: you might add the players found with the WHO command to such a window and select if you want to highlight their sentences in another user window or not (you can do this by adding some spaces in front of the name and manipulating them when the '<OnWinClick' alias is received);
- highlight what every member of your group says;
- much much more... (let me know about your ideas :-)).

Here is an alias that helps showing what every member of your group says:

Name:	Group chat
Pattern:	group *
Hypothesis:	the parameter ('*') contains the name of the character. the string variable *@t* contains the message issued by the player. You have to trap it with a trigger. If you like, you can define *@t* as a local string variable in the body of the trigger.

@NLOCAL w	define a local variable to store the *id* of the user window
@SLOCAL h	define a local variable that contains what

	to put at the beginning of the line (used to indent and to select colors). Let's call this variable *line header*.
@h=@JUST("%1t",9,1)+"-#27[53m"	set the first *line header* to the left justified name of the player followed by a '-' sign and the color for the body of the message
@w=@WOPEQ("Group Chat")	open the window, but if it already exists, then re-use it
@WHILE @t<>""	this @WHILE-@ENDW loop is used to break the message in smaller parts, to make it fit in a smaller window
*@SKIP (@LEN(@t)>40)*3*	if the length of the message now exceeds 40 chars, then skip the following 3 commands
@WADD @w,@h+@t	output to the user window the *line header* followed by the message
@t=""	the message has been completely shown. By setting it to "" causes the loop to end
@SKIP 2	skip the following 2 commands
@WADD @w,@h+@COPY(@t,1,40)	output to the user window the *line header* followed by the first 40 characters in the message
@T=@COPY(@t,41,255)	remove the first 40 characters from the message
@h=" #27[53m"	this line is executed in both the previous cases. After the first *line header*, the remaining ones will always contain spaces and the color for the body of the message
@ENDW	the end of the loop :-)

See also, Advanced programming.

Using mouse

Special actions can be linked to the mouse. There are two different behaviours depending whether a button is pressed over the zone dedicated to the text received from the MUD or on a user defined button.

BUTTON PRESSED OVER THE SCROLL-BUFFER

One can select the variable where to store the word UNDER the mouse.
Furthermore, a sequence of commands can be executed AFTER such assign.
Here is an example:

- select to store the word in the variable @TempS
- the command might be 'JUNK @TempS'

In this way it is very easy to junk useless things.

See also, Using mouse, Mouse and scroll-buffer.

BUTTON PRESSED ON USER BUTTONS

One can define user BUTTONS linked to user defined commands.

Using both behaviours leads to powerful mouse usage.
For example, one can store in the variable @ENEMY the word under the mouse when the LEFT MOUSE BUTTON is pressed.
By defining user BUTTONS with commands like "*BASH @enemy*" and so on, it is very easy to act only by mouse.

IMPORTANT: the commands linked to the mouse can be AUTO-EXECUTED. In such a case, the input buffer is left untouched.
This means that one can have a command ready to be sent in the input buffer and be able to send another command by using the mouse.

User buttons can be enabled/disabled individually.
Disabled buttons disappear from the pad.
If there are too many buttons on a pad, the exceeding ones are disabled automatically.

See also, Using mouse.

Configuration

@BEEP FILE

The sound to be played when the meta-command @BEEP is issued.

See also, Configuration.

BEEP WHEN HIT

The sound to be played when the value of HP found in the prompt decreases.

See also, Configuration.

EXITS TEST

When playing MUD derived from DIKU (like ShadowDale and others), the exits are shown preceded by some text. If set, for example, to "Exits:*", when ELF receives "Exits: north east down" from the MUD, "north east down" will be extracted and shown in the relative panel on top of the scroll-buffer.

When ELF will recognise the line containing the exits, the line right before it will be also shown in the former panel. This is useful in DIKUmuds when moving in BRIEF MODE ON, because that line contains the short description of the room in which you are.

See also, Configuration.

KEYS MODE

You can select how the keyboard is redefined by ELF.
In "directions" mode, moving is privileged.
In "editing" mode, editing is privileged.
A table somewhere else in this manual show the keys definition.

See also, Configuration.

MESSAGES DURING DATA RECEIVE

When receiving data from the MUD, ELF will process a line at a time. After every line, ELF will process Windows Messages to allow cooperative multitasking.

In this way, ELF slows down, but allows smooth operations.

On very slow systems it might be useful to raise this value.

Have some tests, but it is preferable to change the REFRESH RATE instead.

See also, Configuration.

REFRESH RATE

ELF will update the screen only when (almost) idle.

When a lot of lines are received at once this speeds up operations reducing scrolling (which is a slow operation).

Anyhow, if the number of lines received is very high, then the visual result might be ugly.

Setting REFRESH RATE to a value, instructs ELF to update the screen at least every # of lines received.

A good value for this parameter seems to be between 5 and 15, but depends highly on your system and your tastes :-)

See also, Configuration.

USE ALT KEY FOR MENU

Usually the ALT key is trapped by ELF to execute macros.

If you decide not to define macros containing the ALT key and prefer to use the ALT keys for the standard Windows purpose, check this option on.

See also, Configuration.

LOCAL ECHO

ELF is something like a TELNET client. The local echo can be handled both locally and remotely.

It is difficult to explain all the possibile combinations of ECHO that can arise from the TELNET commands.

If you think to have problems with ECHO, try changing this option :-)

See also, Configuration.

TELNET ECHO

The TELNET protocol has some hidden commands. Some of them relate to ECHO handling. The server can decide to enable/disable echo. If something goes wrong (garbage over the link...) it is possible that the wrong state remains on.

With this option you can recover from this.

See also, Configuration.

EXTENDED @INGROUP

Older versions of ELF used a binary logic for the function @INGROUP("member"). If *member* was really a member of your group, @INGROUP returned 1, 0 otherwise.

If you choose to use EXTENDED SYNTAX, then 0 will be returned if not a member, but, instead of 1, the position in the group will be returned if it is a member.

See also, Configuration.

DEFAULT PC NAME

The name of the default Playing Character to be used by ELF.
Click over the edit box with the mouse to easily select the PC to be used.

See also, Configuration, Default PC extension.

DEFAULT PC EXTENSION

You can define (if you are a registered user) several configurations containing different triggers, macros, aliases, variables. One for each Playing Character you have.

These configurations are stored by ELF in files having specific extensions.

Choosing an extension to use, actually chooses the default configuration to be used by ELF at startup.

Click over the edit box with the mouse to easily select the PC to be used.

See also, Configuration, Default PC name.

HISTORY STRATEGY

This option changes the behaviour of the history buffer.

I prefer the *sequence* mode, which is useful when repeating a long series of commands.

Others prefer the *simple* mode.

Try them out and decide :-)))

See also, Configuration.

EXIT CONFIRMATIONS

When leaving ELF you can choose if ELF will ask you if you really want to quit and/or edit untested translations.

See also, Configuration.

LINES IN SCROLL-BACK BUFFER

Sets the maximum number of lines used for the scroll-back buffer. The highest value depends on your system and if you are registered or not :-)

If you set too high a value, ELF will automatically reduce it if any problem arises.

See also, Configuration.

LOG AUTO-SAVE INTERVAL

If you are a registered users you can open a LOG file to store whatever arrives to you from the MUD.

Like any other file, the LOG file is not actually saved to disk by Windows until it is closed. This is the reason why a system crash might lead to a data loss.

This option lets you specify after how many seconds ELF will close and re-open the log file, thus reducing data loss :-)

See also, Configuration.

MAX LINE WIDTH

ELF will split long lines in the scroll-buffer to fit this maximum line length. Colors are preserved.

See also, Configuration.

PROMPT and TICK recognition

See also

PROMPT and TICK recognition

What is a TICK

Time on a MUD passes in a discrete way.
Every some seconds, the MUD recalculates several variables.
We name such a recalculation TICK.
When a TICK arrives, the MUD checks things like:

- hungriness
- thirstiness
- the effects of poisons
- if some spells have expired
- Health Point regain
- MANA regain.

Let's focus on MANA.

MANA is the mental power needed to cast spells and other (few) special operations.

MANA regain happens at the end of a TICK and depends on what the playing character is doing. When SLEEPING or RESTING, MANA regain is higher. Sleeping, one can achieve the best MANA regain.

It is important to note that if one sleeps for a whole minute and wakes up just before tick's arrival, sleeping was completely useless!

It is enough to sleep *around the tick* (right before and a little bit after).

ELF, if correctly set up, is able to tell WHEN the tick will be, allowing one to sleep as little as possible, thus wasting less time.

See also, PROMPT and TICK recognition.

The way ELF recognises the TICK

With the PROMPT TEST, ELF is able to recognise the PROMPT sent by the MUD.

Correctly defining the VARIABLE PART to be analysed, ELF is able to isolate the numeric value of interest.

Analysing those values every time a prompt arrives from the mud, ELF recognises when they increase, thus recognising a TICK.

Advanced tick recognition lets you fine tune this behaviour, to avoid that some spells or special situations that might increase that value artificially lead to a wrong tick recognition.

For example: gaining too much movement (for a refresh spell) would lead to a tick recognition. It wouldn't if we set an upper limit to movement gaining. The same applies to mana or healt points (if we decided to monitorise one of them).

There is a problem if the TICK arrives when MANA (or healt points or movement) is at its maximum.

To avoid it, ELF comes with a trigger named "*lower MANA*" that must be set so that it wastes MANA (or movement or healt points(!)) when it is at its maximum.

As an added security, ELF informs you when the maximum value is reached for the value monitorised with a message (but you can disable this feature). The maximum values are guessed by elf automatically and will work mostly. Advanced players that use different equipment from time to time might have varying values... At present there is no easy way to work around this... Anyhow, expert players will, probably, be expert enough to program some triggers to avoid this problem.

Use the PROMPT WIZARD in the GENERAL CONFIGURATION page for easily setup prompt recognition. Not only you'll enable tick recognition, but you'll have @MYMANA, @MYMOV and @MYHP automatically updated by ELF itself.

See also, PROMPT and TICK recognition.

What happens when TICK arrives

By changing TICK ALERT (SECONDS) one can define how many seconds before TICK's arrival, ELF must execute *@CALL @PreTick*.

5 seconds AFTER TICK's arrival, ELF executes *@CALL @PostTick*.

If *@PreTick* and/or *@PostTick* don't exits, the involved @CALL isn't executed.

See also, PROMPT and TICK recognition.

Aliases used by ELF

ELF defines some aliases for internal use.

<OnConnect	called when the connection with a MUD has been established. You might use this alias to initialise some variables or to enable some triggers.
<OnDisconnect	called when the connection with a MUD has been lost. You might use this alias to disable some triggers or, if you like, to automatically reconnect to the last mud you were connected to (with @CONNECT @LASTMUD). Please note that if decide to automatically reconnect, reconnection will take place even when you wanted to close the connection. It might be better to define a button to be used to reconnect.
<OnWinClick window-id,x,y	called when the left mouse button is pressed over the user window identified by window-id. X and Y are the coordinates of the mouse inside the window. The coordinates are 0-based and scaled by the font size used in the window. The Y coordinate varies accordingly with the scroller (if one).
<OnWinRClick window-id,x,y	called when the right mouse button is pressed over the user window identified by window-id. X and Y are the coordinates of the mouse inside the window. The coordinates are 0-based and scaled by the font size used in the window. The Y coordinate varies accordingly with the scroller (if one).
<OnQuit	called when ELF is about to shut down.

File programming basics

ELF lets you program aliases, triggers, macros, with a powerful language. A powerful language must be able to handle files. ELF's programming language does.

To ease file handling, a few minor assumptions have been made.

The files are line oriented, that means that you can read or write a line at a time.

A line can contain a string or a numeric value.

You can have several files open at the same time, but the number of simultaneously opened files is not unlimited, so you have to keep track of them and remember to close them.

There are meta commands and functions that allow you to open a file for reading, writing or appending, to write a line of text, to read a number or a string and to test the end of file condition.

Variables used by ELF

ELF uses some variables for internal use. Such variables are @CALLed in the following cases:

@AutoExec	right after program initialization
@AutoPC	right after having loaded a new Playing Character configuration
@IdleProc	when ELF has not much else to do :-)
@PostTick	5 seconds after tick detection (the delay is useful if you were sleeping when the tick expired; in this way you are free to wake up, eventually with a trigger :-))
@PreTick	the number of seconds, that you specified in the configuration, before the estimated tick arrives
@QuitPC	right before loading a new Playing Character configuration

What happens when you register

- removed the 500 lines limit in SCROLL-BACK
- fully functional TRIGGER and ANSI wizard
- FIND in the SCROLL-BACK is enabled
- you will be able to define different Playing Characters' configurations
- you will be able to define several MUD connections
- the COMPASS will show 3 user-definable buttons.
- the running title stops and becomes personalised
- the running logo can be stopped
- the HOW TO button disappears
- you will be able to run multiple instances of ELF
- a message shows additional infos

ELF is NOT FREEWARE!
ELF is SHAREWARE :-)

ELF is the result of several months of playing, programming and testing, so I think it is honest for me to ask for a reward :-) 

By registering you'll get several bonuses :-)

The registration fee is **30 US DOLLARS**.

VISA, MASTERCARD, American Express, US Checks, Invoice orders are done through KAGI. Registration is achieved via a small program (REGISTER.EXE) included in the distribution package that will create the actual order form (using encrypted data). You should send such form to KAGI through fax, mail or E-mail.

Another method is using the on-line registration (**VISA, MASTERCARD, American Express**) form by connecting to **http://order.kagi.com/?D6**.

If you like, you can also use snail-mail. Snail-mail registrations can be sent directly to the address below:

Alfredo Milani-Comparetti
Via Velino, 24
60100 - Ancona
Italy

If you use snail-mail, please include cash in a thick envelope. Send cash in US dollars. Remember to include your internet address.

When you have done with the money, send me an e-mail to **MC3078@MCLINK.IT**, specifying that you want to register ELFxWINSOCK with the following data:

- your full name
- your address
- the version you are going to register (this is only for statistical purposes)
- your favourite nickname
- your favourite number (up to 9 digits)
- your E-MAIL address
- how you paid the registration fee

Notes...

As soon as a I will receive your payement, I will send to you an E-MAIL with a small file (the registration data). You simply have to put that file in ELFxWINSOCK's working directory. Nothing more, nothing less :-) As soon as the registration file is in place, you'll get several bonuses.

The version number I ask you to send to me is only for statistical purposes. You will be entitled to use ELFxWINSOCK forever (read README.DOC for more information about this).

Your favourite number will be used, if possible, as a base number for your registration code :-)

The registration will be used by *every new version you will download*, so you will *never have to wait* for me to send an updated version: just pick it up and use it :-)

IMPORTANT!!!

By registering ELF you will be entitled to use ELF on *one computer at a time*. Only one user will be allowed to use it at the same time. You will be allowed to open as many sessions as needed, but you'll have to be *the only user*.

If you plan to use ELF in a large structure, you'll have to register several copies. In

such a case, please e-mail me. We'll find an agreement :-)

Here is a sample registration form:

ELF x WINDOWS's registration form

I have sent to you the necessary money to register ELF x WINSOCK in the following form:

My full name is:
My address is:

The version I am using right now is:
My favourite nickname is:
My favourite (up to 9 digits) number is:
My E-MAIL address(es) is (are):

Notes and suggestions: