Fame

The Reference for Customization

Introduction

Do you know one single man who has ever read an introduction to some reference? Neither do I. So please just hit Page Down.

Configuring Notepad++

It is very convenient to have a dedicated text editor with syntax highlighting and other cool stuff to create and maintain scripts. One example of such an editor is Notepad++. Its quite fast, stable and full of useful features. Fame comes with a set of syntax highlighting files especially for Notepad++, so that you receive:

- syntax coloring
- code completion and parameter hints for all U functions
- quick compilation/syntax check
- colored compilation error report with line numbers and double-click support

Syntax coloring

In the **Tools** directory you will find two files, **u.xml** and **u_udl.xml**. The first one contains U functions, their parameters and short descriptions. The second one contains keywords and some other syntax information. You need to copy **u.xml** to **Notepad++/plugins/APIs.** Then start Notepad++ and select "View → User-Defined Dialogue...". Click the "Import" button and browse to the **u_udl.xml.** This will load U syntax rules. You should now have the U entry in the Language menu.

If you want the code completion and function parameters hint features, you must enable them (Settings → Preferences → Auto-Completion). The recommended configuration is: "Enable auto-completion on each input", "Function completion" and "Function parameters hint on input". You can also change the "From nth character" setting to 2.

Compiling scripts

If you want to be able to compile scripts, you need to install a plug-in. It is called **NppExec**. As you are done with the installation (just copy NppExec's DLL file to the Notepad++'s directory "plugins" and restart the program), you should be able to access the NppExec menu (in "Plugins") and choose "Execute". The command to enter in the dialog box is:

```
cmd /c chdir c:\fame && c:\fame\fame.exe "$(FULL_CURRENT_PATH)"
```

Of course you might want to provide different path to Fame. You can save the command to be

invoked later. Let's save it as “Execute Fame”. It's nice to have a keyboard shortcut, so you can go to “Plugins → NppExec → Advanced Options”. In the “Menu items” section, you can add a new item, let's call it “Fame”. Then select a command you have just entered (Execute Fame). Now go to “Settings → Shortcut Mapper” (in Notepad++'s menu) and select the “Plugin commands” tab. Find “Fame”, double-click it, and assign your favorite compilation key shortcut to it. Restart Notepad++. Now you should see your new command “Fame” in the “Plugins → NppExec” menu. Please note that the key might have been already mapped to some other command (most likely a native Notepad++ feature – there are so many of them) – better check it twice!

Handling compilation errors

Last but not least, if an error is found in a script, you probably wish to know the exact place where the error occurred. This can be done in “Plugins → NppExec → Console Output Filters”. Place a check mark next to a free slot and type in:

```
%LINE%: error*
```

You can also change font color (on the right) to 0xFF, 0x00, 0x00 (red) and make it bold, so that error messages differ from regular output. Now, as the game engine encounters an error during a syntax check, it will appear in the Notepad++ console in red. You can even double-click the error message and you are immediately transferred to the line that contains the error.

One more thing you can do for comfortable work with U scripts is automatic removal of certain output printouts. Switch to the 'Replace' tab in the 'Console Output Filters' window and add entries for any printout that you feel is unnecessary with NppExec, such as copyright lines or memory leak detection status.

Now you have a complete IDE for developing Fame scripts! :-)

U Scripting Language

Plenty of things in the game (including conversations with NPCs) are set up using a scripting language called U. It is not necessary to be good at any programming language to make mods. You can only use a small subset of its features to create simple quests. Using U can extend your possibilities greatly, though.

At the moment, U is interpreted, not compiled. This means that it's terribly slow, but you don't have to worry about the compilation stuff. Actually, compilation IS supported, but executing compiled code is not. Nevertheless, you can make an EXE file out of your script (and it's not a script injected into interpreter's executable – it's really compiled!). This feature is as pretty as it's useless ;-).

You can write your scripts in any text editor and use the game engine to execute them, as well as to just test its correctness. Please refer to the manual for details on the engine's command line.

Several character encoding types can be used, but UTF-8 without BOM is recommended, as it is widely accepted as a portable encoding type.

Script types

There are four major script types in Fame. They all share the same *.u extension and have the same syntax, but they are executed in slightly different manners. The types are:

- NPC scripts
- Global declarations
- General purpose scripts
- Trade profiles
- Location scripts
- Item scripts
- The Handler
- The OnLoad Script

The first (and "the most popular") group contains dialog scripts. They handle chatting with NPC's and all the stuff it involves (quests!). They also may contain some adjustments of NPC's parameters (the ones that cannot be set in monsters.ini), e. g. trade profiles. These are the only files that allow defining actions. What is more, statements outside actions are not allowed here. The only exception are variable declarations, which can be placed anywhere in a NPC script. For details, see the 'Writing dialog scripts' chapter.

The 'Global declarations' group contains declarations of global variables only. Currently, there is only one file in this group; it is the _global.u file. The variables declared in that file are available in ALL other script files. The file name is preceded by the underscore character to make it appear on top of the list when you display it in your file manager. This is very important file.

The 'General purpose scripts' group are 'usual' scripts. They have no constrains about the contents; they may contain almost everything (except actions). Currently, there are no files in this group. As the game develops, there will be probably much more files of the general purpose group.

The fifth group are trade profiles. They make it possible for the hero to trade with NPCs. See 'Writing Trade Profiles' for more details about this group.

The next group are location scripts. They consist of two predefined callback functions, `OnEnter` and `OnExit`. The `OnEnter` function is called every time the hero enters the locations, and the `OnExit` function is called when they leave the location. You can use the functions from the DiffXXX group to obtain last visit time and perform some action based on the time difference (for example: delete some of the location monsters if 7 days have passed).

Item scripts are invoked when items are used. The exact action depends on item category. Currently, only one item category uses scripts (readable items; they invoke their scripts as they are read).

Handler is a special script that groups together various callback functions. It has been designed to be used primarily with tutorials (which usually display a pop up window in response to some user action, thus a callback is needed), but can be used for other purposes as well. The callbacks currently implemented are:

- `OnNewGameStart` – called when a new game starts
- `OnChatEnd (MonType)` – called when the hero ends a talk with somebody, MonType

specifies NPC type index

- `OnEnterLoc (LocWorldX, LocWorldY)` – called the hero enters a location, LocWorldX and LocWorldX specify the world coordinates of the location
- `OnKill (MonsterID, PosX, PosY)` – called when the hero kills a monster or NPC, MonsterID specifies monster or NPC ID (that can be obtained by a call to `AddMonster`)
- `OnPickUpItem (ItemType, Param)` – called when the hero picks up or equips an item, ItemType specifies item type; Param is inventory ID of the slot where the item has been placed, when it's less than 17, the item is currently being equipped

The last group is the OnLoad script. There can be only one OnLoad script in the game and it appears as the _onload.u file. Its role is to initialize the game; it should load the world file, select a default location, set hero's attributes, etc.

The OnLoad script contains one important callback function, `OnChooseWeaponSkill.`

```
function OnChooseWeaponSkill (SkillID)
```

The function is called when the player confirms creation of a new character and it is time to equip him with weapons appropriate for the chosen weapon skill.

Language basics

General information

The U language is NOT case sensitive. You can type its statements in lowercase, UPPERCASE, camelCase, even in TeEnIeBlOgGiEcAsE, just as you wish to.

Simple commands

There are few commands you can use primarily for debug purposes, for example to print text in the console window (if it is present, of course). They act much like usual procedures, but they do not require parentheses for their argument list. You can separate arguments from the command name by almost any character (usually it is space, though) and you can separate one argument from another by any non-digit and non-letter character (but usually it is comma character). Just look at the following examples.

The Print and Println commands

The Print command, as you probably expect, prints text on the screen (i. e. in the console window).

syntax:
```
Print <text>
Println <text>
```

These commands only work when console window is present and it exists mainly for debugging purposes.

The Input command

The Input command allows you to read keyboard input:

Syntax:
```
Input <variable>
```

User must hit Enter key to end typing. All entered white-space characters are skipped (i. e. if you enter several words separated by spaces, only the first word will be actually read into the variable). This command only works when console window is present and it exists mainly for debugging purposes.

The System command

The System command allows script to execute system commands. What exactly you can do with System command, it depends on the operating system you are using.

Syntax:

```
System <command>
```

Example:

```
system "dir"
```

This example will execute the `dir` command (listing files in the current directory).
This command exists mainly for debugging purposes.

The Incr and Decr commands

The `incr` keyword is an abbreviation for 'increment'. It just adds 1 to the specified variable, e. g.:

```
var my_var = 5
incr my_var
```

...makes `my_var` evaluate to 6. The `decr` command, as you guess, does the opposite.

The Exit command

The `exit` command ends execution of the script immediately.

Comments

All the characters following the '//' sequence (up to the line-break) are treated as comments (just like in C++). You can enter anything you want there and it will not affect your program.

There are no block comments in U (unlike C++, where we do have the slash-asterisk and asterisk-slash sequences). You can only use multiple '//'.

Example:

```
print "This line will execute."
print "This one too."
//print "But this one will not. It's commented out."
print "And this one will be executed normally..."
```

Variables

The variables can be declared with the `var` keyword. There are no variable types in U. You can only store integer numbers in variables. More specifically, an unsigned 64-bit value is reserved for each variable. However it is not recommended to use values larger than `INT_MAX-1`.

A variable declaration looks like this:

```
var MyVariable 0
```

or equivalently (and maybe more familiar to programmers):

```
var MyVariable = 0
```

Please note that both leading and trailing space is required when using the '=' character here (unlike C/C++, where the spaces are optional and much unlike Bash programming, where they make a syntax error!).

The character '0' at the end of the declaration is a value which we assign to the variable immediately after its creation (i. e. we initialize the variable with the value).

Naming of variables must conform to some rules:

- first character must be an alphabetic character (A-Z)
- the rest of characters must be alphabetic characters or underscore characters ('_')
- underscore as the first character of a name should never be used, it is reserved for internal use

So, the following names are valid:

```
var MyVariable1 0
var My_variable 0
```

...while the following are not:

```
var 1MyVariable 0 // error!
var _MyVariable 0 // error!
```

Please note that variable names are NOT case sensitive (unlike the C++, and much unlike the Linux!).

Depending on scope, we can distinguish three types of variables:

- global variables
- local variables (module scope)
- local variables (block scope)

The first kind of variable is easiest to distinguish, because every global variable is declared in the _global.u file, which we have already discussed of.

Local variables can be declared either outside or inside a block of code. We will know more details

about it as soon as we talk about functions. At the moment we can only say that all local variables (both "module" ones and "block" ones) are only "visible" in a file they are declared in.

You can assign a value to a variable at any time:

```
var a 0
println "Variable declared."
a = 5
println "Variable value changed."
```

It is not possible to initialize variable to anything but a bare number. However, it is possible to assign it a value returned from a function. The `ValueOf` function can be used to obtain a value of another variable, e.g.:

```
var a = 1
var b = 0
b = ValueOf (a)
```

The language basically does not differentiate between various white-space types, so that you can do assignments and other operations on the same line as initialization if you separate these operations by spaces:

```
var dx = 0 dx = ValueOf(rx)   incr dx // in C++ we'd just say 'dx =
rx + 1'
```

This partially compensates the syntax restrictions for variable initialization.

You can use the `print` and `println` commands to write variable's value to the output:

```
var a = 1
println a
```

Conditionals

The basics of conditional expressions in U

As in almost every programming language, in U you can check conditions too. And as in almost every programming language, you have the 'if' statement. It works much like its "cousin" from C/C++, meaning that it can accept complex conditions like:

```
if (MyVar == 33 && MySecondOne != MyThirdOne)
{
 println "Condition is true."
}

if (MyVar == 33 && !MyOtherVar)
{
 println "This condition is also true."
}
```

As in the C/C++, variables are implicitly "converted" to Boolean type in logical expressions. More precisely (as there's actually no Boolean type in the U), all non-zero values are evaluated as 'true' and zeros are evaluated as 'false'.

The difference between the C/C++ and the U is that you MUST follow the 'if' statement by '{' and '}' characters, even if there is only one command in the block:

```
if (cnd)
{
 println "Something"
}
```

There are no C-like shortcuts like:

```
if (cnd)
 println "Something" // syntax error in U!
```

Note that nested constructs with parentheses are too difficult for the parser, so that this snippet, for instance, will not compile:

```
if (a && (c || b)) // syntax error
{
 println "Doh."
}
```

The 'else' block

The `else` block can be used together with the `if` statement:

```
if (cnd)
```

```
{
 println "Something"
}
else
{
 println "Something else"
}
```

We have talked about the rules you must follow when creating an `if` block. The same rules apply for the `else` block.

The `else` and `if` statements can be used together to form a more complex language constructs like:

```
if (cnd)
{
 println "Something"
}
else if (cnd2)
{
 println "Something else"
}
else
{
 println "Something different"
}
```

Operators

The operators you can use to construct conditions are very similar to the ones knows from C/C++. They have the same symbols (with one small exception), the same priority and other features. The available operators are:

Operator name	Symbol	Quick example
Equal	`==`	`var1 == var2`
Not equal	`!=`	`var1 != var2`
Not	`!`	`!var1`
And	`&&`	`var1 && var2`
Or	`||`	`var1 || var2`
Xor (eXclusive or)	`^^`	`var1 ^^ var2`

As you probably know, in C/C++ there's no operator like '^^'. But in U there are no bit-wise operators (like '&' or '|' in C/C++), so the xor operator has been assigned the '^^' symbol, just to make it look more like the other logical operators ('&&', '||'). In C/C++ you have the '^' operator, which does exactly the same.

Loops

Loops are constructs that simplify repetitive tasks greatly in common programming languages, although in U they are rarely used. Nevertheless, they do exist. The syntax is Basic-like, example:

```
for i = 0 to 10
{
 println (“something”)
}
```

There is no way to change step value, so loop counter is always incremented by one.

Additional control over loops is provided by keywords: `break` and `continue`. The `break` statement exits the loop immediately, while the `continue` keyword skips all the statements following it and begins a new iteration. Similar behavior can be observed in languages like C, C++ or Java.

If the variable used by a loop is a global variable or module local variable, the variable name is hidden (like in C++). Block local variables cannot be hidden, so it's an error to use them as a loop counter. You can also use a non-existent variable name as a counter, it will be created automatically in this case.

The range can be specified using variables, but more complex expressions are disallowed:

```
for i = a to b // OK
for i = a to a + 10 // not OK!
```

Functions

Function is a set of commands with assigned name and sometimes with a parameter list, returning some value as it ends its execution.

In U, functions have some differences when compared to popular programming languages like C++ or Pascal. Here they are:

- every function must return a value, but it can return implicitly
- when using an implicit return, a value of 0 is returned
- you must return a constant value or nothing (which equals returning 0), you cannot return a variable
- only an unsigned integer values can be returned
- only an unsigned integer values can be used as parameters (although some built-in functions can also have string and float parameter types)
- there are no default values for parameters
- parameter values are always passed by value, there are no references nor pointers
- you can only use a returned value to:
  - assign it to a variable
  - use it in a conditional expression
- you cannot overload functions
- there are no inline functions
- there is only one, default calling convention

Functions in U are similar to C++ in that:

- functions with no parameters are allowed (although you still must include parentheses)
- you can use parameters as if they were local variables (you can change their value too)
- you are not obliged to use function's return value (just like in C++)
- you can recursively call functions (i. e. you can call a `MyFunc` function from inside of `MyFunc`)

Unlike C++ or Pascal, you need not to declare functions. Once you have a definition (i. e. function's body), you also have the function declared. You do not even have to place the definition on the top of the file. It can be almost anywhere in the file and it will be "visible" to any call.

Syntax:

```
function <name> (<parameter_list>)
{
 <commands>
 [<return_statement>]
}
```

You do not specify function's type nor parameters' type, because you only use one type (unsigned integer). Instead you use the "function" keyword to declare a function. You must include parentheses even if you do not have any parameters.

You can place any number of return statements. The compiler will warn you if you do not have one, but it will not warn when not all control paths would return a value! It just checks whether any return command exists in the function's body.

Example:

```
function seven()
{
 return 7
}
```

The following example shows how to use parameters:

```
function test (four, nine)
{
 if (four == 4 && nine == 9)
 {
  println "Everything is fine."
  return 1
 }
 else
 {
  println "Something is wrong..."
  return 0
 }
}
```

You can use the returned value in the following ways:

```
test (4, 9) // do not use the returned value at all

if (test(4, 9)) // use it in a conditional expression
{
 println "Function succeeded."
}

var a = test (4, 9)
```

The following example shows an implicit return. If `arg` is equal to 10, we return a value of 1 explicitly, but if `arg` is something else, we do not have the `return` statement at all, which means that the function returns automatically and the return value is 0:

```
function Fun (arg)
{
 if (arg == 10)
 {
  println "Arg is ten."
  return 1
 }
 else
 {
  println "Arg is not ten."
 }
}
```

We can also use “naked” `return` statement, i. e. without a value:

```
function Fun (arg)
{
 if (arg == 10)
 {
  println "Arg is ten."
  return
 }

 return 1
}
```

For a list of predefined functions, see the "Predefined functions" chapter.

Labels

Yes, the infamous 'goto' command is available too. Although in modern programming languages like C++ it is generally not recommended to use 'goto', in game scripting languages like U it can prove to be very useful, as the U doesn't have many other control statements (like 'break' or 'continue' in C++).

You declare and use labels as follows:

```
if (Cnd1)
{
   if (Cnd2)
   }
    goto MyLabel
   }
   println "This line will be skipped if Cnd2 is nonzero."
}


MyLabel:
 println "Here we are!"
```

You may use 'goto' almost anywhere in code, but there are some restrictions connected with actions, which we are going to discuss later.

Inheritance

The inheritance mechanism implementation may be incomplete and thus it is not recommended to use it.

U language allows you to inherit things from other scripts you or somebody else has already written. There is one important limitation: it can only create flat inheritance hierarchy (script B can inherit from A and C can inherit from B at the same moment but C won't automatically inherit from A then).

You indicate that script is derived from another script by adding the `extends` keyword.

Syntax:
`extends <script_file_name>`

This syntax may lead to some confusion, suggesting that the whole script file is inherited. This is only partially true, as only functions and variables are actually inherited. Any other elements (procedures, actions and so on, also any elements outside function blocks) are ignored.

We will refer to the script given by the script_file_name parameter as to "base script". We will also refer to the script having the `extends` keyword as "derived script".

The script file name must not include a path. Base script must reside in the same directory as the derived script.

Inheriting functions

When a function is called from the derived script, the U interpreter searches the derived scripts and if it finds a function of the given name, it gets called. If such function is not found, then the base script is searched and matching function is called (if found).

If you have a function `MyFunction` in your derived script and it also exists in the base script, you should precede the `MyFunction` definition in the derived script by the keyword `overridden`. You can call the base `MyFunction` from the overridden one, using the keyword `super`.

Note: `super` may not work correctly in the current version of the engine. Use with care or better avoid.

Example:

[scriptbase.u]

```
function MyFunction (arg1, arg2)
{
 println "Base script here!"
}
```

[derived.u]

```
extends "scriptbase.u"

overridden function MyFunction (arg1, arg2)
{
 println "Derived script here!"
 super // let the base script do its job too
}
```

In the above example both texts will be written in the console window, because we have used the keyword `super`, making a call to the base function.

Please note that the derived version of function must have the same number of parameters as the base version. The parameter names may differ.

Inheriting variables

You can refer to variables declared in the base script from the derived script. However, if you declare a variable named `MyVar` in the derived script and you had declared a variable of the same name `MyVar` in the base script, only the one from the base script will be used whenever you refer to the `MyVar` name. Therefore you should always keep in mind the list of variables from the base script and avoid using the names from the list when declaring new variables in the derived script.

Writing dialog scripts

Dialog designing is based on actions. Every NPC has its own (and only one) dialog script (that is, an *.u file) containing a set of actions. An action is an atomic unit of dialog. One action can be used in one or more different dialogs with a certain NPC. Also, a set of actions (and/or their order) used in a certain dialog can change during play time, if the script writer wishes to.

Actions

Actions are constructions similar to procedures. They are sets of commands with an assigned name. There are several major differences, though:

- you cannot nest actions like you do with procedures
- there's no way to call actions directly from outside an other action; they can only be called from the 'option' command (only if user selects the option enumerated by that command); it can only be placed in an action block one (and only one!) action is always preceded by the 'default' keyword; it marks a default action (the one that starts up the dialog)
- you can (but don't have to) initialize the 'text' property of the current action; its value is displayed on screen and can be seen by the player as the NPC's talk. You can set this property more than once per single action; only the last assign counts. If you do not set 'text' at all, the action's commands shall be normally executed, but in that case you should make sure that you leave the action properly (by 'end', 'trade', or 'shift' command), as the dialog options shall not be displayed on screen, even if they have been enumerated
- you can (but don't have to) place one or more 'option' commands in the current action to give the player a number of choices. If you don't enumerate any dialog option, the dialog shall end immediately (but action text, if present, shall be displayed).
- you cannot use 'goto' command from inside an action
- you cannot place labels in an action (i. e. you cannot jump into them by 'goto')
- you cannot place procedure definitions ('def_proc') in an action, but you still can call procedures defined outside

You usually treat an action as a single 'NT-YR' (NPC Talks, You Reply). A single action defines text displayed on screen (this is the NT), and several possibilities of player's replies (these are the YRs).

Another (quite rare) possibility to use an action is a scheme referred to as 'MR' (Multi-Reply). As you can observe, buttons that display player's replies have only limited space for text, so these replies must be as short as possible. That's not fair! NPC can even talk us to death, and we can only say a word or two... Let's stop that! So there's a small trick available and it has already been mentioned; you do not set 'text' property for the action and - voila! - text from the previous action does not disappear from screen, but you are able to speak again!

Control flow in actions

All statements inside an action are executed as they were inside procedures; so does the option statement (it adds a new dialog option to a local list; the list is destroyed as we leave the action block). An assignment of 'text' property works in a similar way. After execution of all action's statements, we enter the modal loop. Then, the dialog window is handled as every window does - the control is delegated to the main game engine. By the moment the player chooses some dialog option, control is returned to the script; we jump into the action selected by player and the scheme continues.

Here's an example of a short conversation:

```
default action hello
{
 text = "Hi, stranger."
 option "Hello.", hello2
 option "Goodbye", end
}

action hello2
{
 text = "I'm actually not supposed to talk to you, so you should
go now."
}
```

This one starts with action 'hello' (because it is default). The NPC says "Hi, stranger" and the hero has two possible replies: "Hello.", which leads to action named 'hello2', and "Goodbye", which ends the conversation. If the player selects the button "Hello.", the action 'hello2' is executed and another text is displayed. Action 'hello2' has no more dialog options, so only one button can be selected now – clicking it just ends conversation.

You may change a default action any time, using the `SetDefaultAction` command:

```
SetDefaultAction hello2
```

Since then, 'hello2' is the default action and conversation with this NPC begins from 'hello2'.

Items

An important thing is that the scripting engine gives you an access to the Player's inventory. You can add items to the inventory or take them from the Player, you can also check if the Player has specified item in the inventory.

To handle items, you use *item definitions*. An item definition is:

```
<item alias> <item name> <quantity> <durability_requirements> <required_flags>
```

Item alias is a name you give to an item to identify it in the current script file. An alias declared in one file is not recognized anywhere else (unless you declare the same alias elsewhere). It also does not lead to any conflict if you declare two aliases with the same name in two different files. Aliases are fully local beings.

Item name is the name of the item provided in the items.ini file.

Quantity – as you can see, quantity is assigned to every item.

Durability_requirements – how damaged the item is allowed to be for the NPC to accept it. Possible values are:

- 0 (not important)
- 2 (50%)
- 3 (75%)
- 4 (100%)

With *durability_requirements* equal to 4, the item must be in a perfect condition, while 3 means that the item must have at least 75% of its 'hit points'. Usually you use the value 0, meaning that any condition is accepted.

The *durability_requirements* value also influences the initial 'hit points' of a newly created items when an NPC gives an item to the player character.

Required flags – What flag this item needs to have to be accepted. Usually you set this to 0 (not important). You may, for example, set it to 256, which is *flgCrafted* flag. Please note that these flags are NOT the same as item definition flags (described in the 'Item Definitions' chapter) – you should never confuse these two!

Flag name	Flag value	Description
flgBSIdentified	1	The item's blessed/cursed status has been revealed.
flgBlessed	2	The item is blessed.
flgCursed	4	The item is cursed.
flgIdentified	8	The item has been identified.
flgBSRandom / *flgAltCorpse*	16	*flgAltCorpse* – the item is a corpse and has an icon other than the default; this flag is mostly used internally and probably has little meaning in scripts *flgBSRandom* – for internal usage only
flgPoisoned	32	The item is poisoned.
flgActivated	64	The item is a tool and its primary function has been

		activated (e.g.: a torch is lit).
flgVirtual	128	Not actually an item (for internal usage only!).
flgCrafted	256	The item has been crafted by player.

An example of item definition:

```
item "The debt", "gold piece", 200, 0, 0
```

What do we have here? It's just 200 gold pieces, identified in the script file by the string alias "The debt". You can now use the definition to check if the Player has at least 200 gold pieces to pay some debt:

```
if (HeroHasItem ("The debt"))
{
 DoSomething()
}
```

...or to check if the Player has ANY gold:

```
if (HeroHasSomeItem ("The debt"))
{
 DoSomething()
}
```

You must put item definitions on the top of the file, before all actions that could potentially use them. Otherwise you will probably get an error message "undefined item" or something like that.

Note: item's material and special data is also matched (by *HeroHasItem* and *HeroHasSomeItem* functions). To disable this, set item's material to 1 (using *SetupItemMaterial*) or item's data to DWORD_MAX (using *SetupItemData*).

Common thing to do with item definitions is to take items from player and to give items to him. For example, we could inform the Player that he has paid his debt, so we can return his jacket to him:

```
if (HeroHasItem ("The debt"))
{
 TakeItem (“The debt”)
 GiveItem (“Hero's jacket”)
}
```

Of course, the “Hero's jacket” string is also an item definition alias.

You can also specify corpses as items, but this is slightly different. As you may have noticed, corpses are not usual items in the sense that they are not defined in the items.XX.ini file. Therefore, writing:

```
item "Goblin's remains", "goblin corpse", 1, 0, 0 // wrong!
```

...will not work. Instead, you can write:

```
item "##Goblin's remains", "goblin", 1, 0, 0
```

Thus, you specify monster's name instead of item's name, and item alias follows two hashes ('#'). Whenever game engine sees item alias beginning with hashes, it assumes that item's name is actually monster's name and it should 'create' a corpse out of it.

Instead of item name, one may specify its number, following a single hash character ('#'). This is a zero-based index of the item definition as it appears in the items.XX.ini file. For instance, the following declaration specifies a short swort, because it's the first entry in the items file:

```
item "a non-magical sword", "#0", 1, 0, 0
```

This is very useful when referencing items that do not have a unique name.

Repairs

Some NPCs would repair your damaged items. To display the repair window (i. e. inventory window in repair mode), you just redirect current action to 'repair', just as you would do with trade window:

```
option "Can you repair my weapons?", repair
```

Repair cost for 1 damage point is the value of item divided by its maximum durability, rounded to the nearest integer and doubled. For example, a broad sword is normally worth 250 gold pieces, so when its durability equals 39/40, then total repair cost is 12 gold pieces. Every NPC has exactly the same way of calculating repair cost (unlike trade prices).

Curing

Some NPCs have medical skills and are able to cure hero's diseases and/or restore their missing limbs. This is done in the similar fashion as item repairs – as a special dialog option redirection:

```
option "Can you cure my diseases?", cure

option "Can you restore my limbs?", cure_limbs
```

Note that cure not only removes all diseases, it also restores Hit Points, as well as it removes bleeding/poisoned status and fixes broken limbs.

Also note that cure_limbs only applies to limbs that have been non-permanently chopped off. It doesn't fix broken limbs nor it restores permanently lost limbs.

Butchery and Cooking

Some NPCs have butchery or cooking skills which can be triggered in similar way to repairing or curing:

```
option "Will you take care of this flesh?", butchery

option "Can you cook this meat?", cooking
```

In case of Butchery, the first appropriate corpse from the hero's inventory is used. The hero also needs some money (5 gold pieces) as a payment. This is done automatically, so you don't have to implement the payment in the script.

In case of Cooking, the first appropriate piece of raw meat from the hero's inventory is used. The hero also needs some money (5 gold pieces per piece of meat) as a payment. This is done automatically as well.

Traveling through locations

Some NPCs would join hero and visit various locations along with him. As they reach a specific destination, the OnEnterLoc function is called in the NPC script (if it is present, of course). You might want to use it to provoke some activity, for example the NPC can comment the new place. It looks like this:

```
function OnEnterLoc (LocWorldX, LocWorldY)
{
    if (LocWorldX == 15 && LocWorldY == 23)
    {
        NPCUnfollowHero()
        SetDefaultAction "hello2"
        incr Quest17Counter
        NPCSetActivity ("Aruos", "#1", 17) // wander
        NPCStartChat ("Aruos", "#1")
    }

    return 0
}
```

The callback function has two parameters, `LocWorldX` and `LocWorldY`, which identifies the location. Usually you want to test the parameters against some criteria. It is quite reasonable for NPC to stop following the hero if they reached their final destination, set new activity type and/or to start a chat.

Dialog script localization

All script examples in this reference assume that there is only one language defined in the game. However, the mod creator has a possibility of create quests that work fine with several different languages. In case of dialog scripts, this is done by writing dialogs in all supported languages at once. Languages can be separated by '|' character:

```
default action hello
{
 text = "Witaj, przyjacielu. Co mogę dla ciebie zrobić?|Welcome,
my friend. What can I do for ya?"

option "Ktoś ty?|Who are you?", who
option "Helo|. Spytać o coś można?|Hello. Can I ask you a
question?", question
option "Pohandlujemy?|Let's trade!", trade
option "Potrafisz naprawiać przedmioty?|Can you repair items?",
repair
option "Spadaj?|Get lost!", end
}
```

In this simple example, we have two languages supported: Polish (first one) and English (second one). This order is determined by an entry in the **Data\game.ini** file, which looks like this:

```
all_available_langs = "pl|en"
```

If you have specified two languages in game.ini, then you MUST supply all dialog texts in two languages. If you forget to do this, the script won't compile. If you don't wish your mod to support more languages, you must change the above game.ini entry.

Dialog Data

There is a mechanism that enabled the script to process some data received dynamically from the game engine. The mechanism, referred to as Dialog Data, or Dialog Data Resource, or DialDataRes, allows you to interact with the randomly generated stuff.

Dialog Data is typically initialized with some data during some kind of random content generation, for example when generating a location. The script should hook to the Dialog Data as soon as the data is ready and retrieve it. The stored data is considered extremely volatile and may get cleared in any moment, so it is important to retrieve it as soon as possible.

Just like in any other element of the U language, there are two data types: numbers and strings. Each piece of data has a unique ID. Note, though, that the nature of Dialog Data is quite ephemeric, so that if you need a number with ID=1 and you attempt to obtain it when it's too late, you may get an exception or silently retrieve some other data, so that one needs to be very careful here.

String data is normally used when setting action text and dialog options (i.e. buttons with choices for the player). You can use the format '$n', where 'n' is an ID of the string that you want to insert. An example of such an action:

```
action a1
```

```
{
     text = "Do you choose $1 or $2?"

     option "$1", choice1
     option "$2", choice2
}
```

Numeric data can be retrieved using the function `GetDialDataInt`. An example:

```
AddTwoArenaMonsters (7, 5, 12, 9)
id1 = GetDialDataInt (3)
id2 = GetDialDataInt (4)
```

As already explained, every set of Dialog Data is available after placing some kind of randomly generated stuff. In the above example two monsters are generated. The function that generates them, `AddTwoArenaMonsters`, adds four Dialog Data values to the storage. First two, with IDs 1 and 2, are string values, while 3 and 4 are integer values. Other functions generating content may set different Dialog Data values. Every time some function is putting a value into the Dialog Data storage, it typically clears previous values, if any, so that you cannot rely on their longevity.

In addition to the values set by functions from U functions, there is one case where values are set elsewhere.
After a partially random location has been generated, four Dialog Data values are set. They are guaranteed to survive at least until `OnPostInit`, so that you can safely retrieve them in that callback. They are:

ID	Type	Contents
1	number	X coordinate of the predefined part of the location
2	number	Y coordinate of the predefined part of the location
5	number	Rightmost coordinate of the predefined part of the location
6	number	Bottommost coordinate of the predefined part of the location

Callback functions

There is a number of predefined function names. If such a function is defined in a script, it may get called by the engine in certain circumstances. Many tasks cannot be accomplished without callbacks.

Location callbacks

Callback	When Called?	Purposes
OnInit	Before the location is loaded or generated.	• RestrictMonsterGeneration • everything else that may affect random

Callback	When Called?	Purposes
		generation
OnPostInit	After the location is loaded or generated.	• Adding monsters to partially random locations • Setting up monster inventories • Setting up guarded cells • Setting up NPC's spells
OnNewSession	After the location is loaded or generated (this happens after OnPostInit and OnEnter) plus every time the game is loaded.	• Adding location effects • Group setup
OnEnter	After the hero enters the location.	
OnExit	Before the hero exits the location.	
OnDestroyItem	Every time an item is destroyed. Parameters: X, Y, Type, Count.	

NPC callbacks

Important note: In order to use NPC callbacks, one needs to ensure that the NPC script has been loaded. Normally it is not unless the hero has already chatted with the NPC. There is a number of other activities that may cause the script to get loaded and cached, but since this happens totally 'behind the scenes', the script author should not rely on this and load the script manually (using the `NpcLoadScript` function, usually in `OnNewSession`) every time he expects some callback to be used.

Callback	Description
OnInit	Called when the NPC script is loaded for the first time. The place to add spells and non-guaranteed items. In both cases, the NPC should have the flgLoadScript flag set.
OnKill	Called when another NPC is killed. Parameters: MonsterID, PosX, PosY. All these parameters refer to the dead NPC.
OnDeath	Called when the NPC dies. Parameters: X, Y (position where the NPC died). If any item dropped by the NPC should be guaranteed, this is the place where they must be added (possibly using AddItem or AddItemEx).
OnGiveItem	Called when the hero gives an item to the NPC. See the chapter 'OnGiveItem' for more info.

Item callbacks

Callback	Description
OnRead	Called when the hero reads the item. Parameter: Type.

OnGiveItem

This is a callback function used to handle giving items to NPCs. The syntax is:

```
function OnGiveItem (Type, FlagIdentified, FlagUnique,
FlagBlessed)
```

The parameters allow the script to decide whether the NPC accepts the item or not and generally what to do with the item.

`Type` – item type
`FlagIdentified` – equal to 1 if the item has been identified, 0 otherwise
`FlagUnique` – equal to 1 if the item is unique (artifact), 0 otherwise
`FlagBlessed` – equal to 2 if the item is blessed, 4 if cursed, 0 otherwise

The return value is important – it tells the engine the "decision" of the NPC. Possible values are:

Value	Meaning
1	Accepted – item has been accepted by the NPC. If this flag is not set, the item is not taken from the player's inventory.
2	Remove – item will be removed from the player's inventory
4	Talk – the NPC will start a dialog with the player
8	Identify – the NPC will attempt to identify the item. This requires special variables - `_Identify` and `_ItemType`.

The values in the above table can be combined, e. g. the value of 13 (8+4+1) means that the item is accepted, the NPC will try to tell something to the player and will also identify the item.

Whenever 8 is returned (most likely combined with other flags), the script is expected to have declared two special variables: `_Identify` and `_ItemType`. Both should be initialized to 0. When `OnGiveItem` returns, `_ItemType` is set to the type of the given item, so that further script calls are able to access and use it (the value of the `Type` parameter in `OnGiveItem` cannot be assigned to other variable in the script because of the scripting language limitations). `_Identify`, on the other hand, is meant to be written by the script rather than read. The script is expected to set

the variable to 1 as soon as the NPC decides that the item will be actually identified (for example, he made sure that the player character has enough money to pay for the service).

OnSacrificeItem

This will be called every time somebody sacrifices an item on an altar. Normally the callback function is placed in the **_global.u** script and there is only one instance of it. The syntax is:

```
function OnSacrificeItem (LocWorldX, LocWorldY, PosX, PosY, ItemType, ItemCount,
DeityID)
```

`LocWorldX, LocWorldY` – coordinates of the location where the altar exists
`PosX, PosY` – altar's cell position in the location
`ItemType` – type of the sacrificed item
`ItemCount` – number of sacrificed items
`DeityID` – ID of the deity assigned to the altar

If more than one item type is sacrificed at time, the function is called several times in a row with different types as a parameter.

The main purpose of the callback is to specify conditions of game endings (see the `HeroGrantChampionStatus` function).

OnSacrificeRareItem

This is called if somebody tries to sacrifice a rare item – that is, an unique item or an item that is never random-generated. The purpose is to prevent the player from sacrificing (and thus destroying) an important item without which completing the game (or a quest) would be impossible or at least very hard. This 'category' includes artifacts, keys and other special items.

The syntax is:

```
function OnSacrificeRareItem (LocWorldX, LocWorldY, PosX, PosY, ItemType,
ItemCount, DeityID)
```

The meaning of the parameters is the same as in the `OnSacrificeItem` callback.

The return value means:

- 0 – do not accept (the item remains)
- 1 – accept (the item disappears)

Dropping items

NPCs will drop items when killed. The recommended way to have NPCs dropping non-important items is to use the `OnPostInit` callback function in a location's script and the `NpcAddInvItem` function to set up NPC's inventory:

```
function OnPostInit()
```

```
{
   NpcAddInvItem ("Gandalf", "#1", "gold piece", 100)
}
```

Please note, however, that NPC's inventory is available during the game, so that the items from it may be stolen, destroyed or lost in any other way. Thus you should consider the inventory as potential, non-guaranteed drop. If you need the NPC to drop a guaranteed item, use the `OnDeath` function instead:

```
function OnDeath (x, y)
{
   AddItemEx ("", "gold piece", 10, x, y)
}
```

The downside of the `OnDeath` approach is, obviously, that the item will not be visible in the monster's inventory until the NPC dies. Some players may object on this, others don't like it when a crucial or unique or very valuable item gets lost. The choice is yours.

Nameless creatures (monsters and common NPCs) can have items added to their inventories as well. You can use the function `CreatureAddMiscItem` for that purpose.

Writing trade profiles

Some NPCs can trade with the hero. If you want NPC to trade, you must do three things:

- create (in dialog script for the NPC) an action leading to 'trade'
- create a trade profile for the NPC
- create a connection between the dialog script and the trade profile

The first thing is easy:

```
action Example
{
 text = "Do you want to trade with me?"
 option "Yeah, sure.", trade
 option "Not now.", end
}
```

You do not define the 'trade' action. Even more: you can't do it, because 'trade' is a reserved word. It doesn't actually name an action. It closes the dialog window and opens the trade window. If the trade is completed, you do not return to the dialog window any more until you [T]alk to the NPC again - please consider this when constructing a dialog!

The second important thing to do is the trade profile itself. It must be placed in a separate *.u file, but you can reuse one trade profile for many different NPCs. It enables you to create several common profiles, e. g. a standard village shopkeeper profile, a city chemist profile, a merchants' guild member profile, a lone traveling merchant profile, etc.

A typical trade profile consists of only one function call. The function is `DefineTradeEntities`. It takes one string parameter, which specifies the details of trader's offer. The string is actually a list of tags. Possible tag values are listed in a table in the 'Settlements' chapter, under the 'treasure' keyword. Additionally you can use the 'item' tag, which specifies a particular item rather than the whole category. Here is an example of a complete trade profile:

```
DefineTradeEntities ("food,item:empty bottle,item:empty half-litre,item:bottle
of liquor,item:strange liquor,item:bottle of water")
```

This line defines a profile for a shopkeeper buying and selling food stuff. The 'food' category is specified collectively, while all additional items (drinks) are listed using the 'item' tag.

To assign a trade profile to the NPC, do something like this (in NPC's dialog script!):

```
LoadTradeProfile ("vlg-blacksmith.u" 2.0, 0.5, 1, 100, 5000)
```

All functions used above are explained further in this document.

Writing location scripts

As already explained, the location scripts typically contain at least two functions, `OnEnter` and `OnExit`. They do not have any standard purpose. You can leave them empty if you want, but you should not attempt to create a location script without one of these functions inside. The location script must be placed in the **/Data/Scripts/Locations** directory (note that this is NOT the same directory as /Data/Locations!) and must have a proper file name. The file name is determined by location file name. For example if the location file is village.map, then the location script should be named village.u. You are not obliged to create the script for every location you have.

Here is an example of a location script for a small village. There is a couple of empty callbacks here. More interesting things happen in the `OnPostInit` function, which is called only once every per game – there we set up NPC spells and items, as well as guarded cells. There is also the `OnNewSession` callback. This one is called every time you load a saved game and additionally when you start a new game. This callback can be used to set up groups:

```
function OnInit()
{
}

function OnPostInit()
{
     NpcAddSpell ("Abilidus", "#1", "Holy Bolt", 30)
     NpcAddSpell ("Abilidus", "#1", "Shield", 30)
     NpcAddSpell ("Abilidus", "#1", "Magic Bolt", 30)

     NpcAddGuardedCell ("eluuny", "Asrigam", "#1", 11, 5)
     NpcAddGuardedCell ("eluuny", "Elumor", "#1", 29, 19)
     NpcAddGuardedCell ("eluuny", "Elidral", "#1", 38, 17)

     var count = 0
     if (Chance (4, 5))
     {
          count = Rnd (10, 40)
          NpcAddInvItem ("Abilidus", "#1", "gold piece", count)
     }
     if (Chance (4, 5))
     {
          count = Rnd (1, 3)
          NpcAddInvItem ("Abilidus", "#1", "potion of focus",
count)
     }
     if (Chance (3, 5))
     {
          count = Rnd (1, 3)
          NpcAddInvItem ("Abilidus", "#1", "potion of healing",
count)
     }
     if (Chance (1, 350))
```

```
     {
          NpcAddInvItem ("Abilidus", "#1", "Spellbook of Holy
Bolt", 1)
     }
}

function OnEnter()
{
}

function OnExit()
{
}

function OnNewSession()
{
     AddGroup ("townspeople_eluuny")

     NpcSetGroup ("eluuny", "Aruos", "#1", "townspeople_eluuny")
     NpcSetGroup ("eluuny", "Abilidus", "#1", "townspeople_eluuny")
     NpcSetGroup ("eluuny", "Asrigam", "#1", "townspeople_eluuny")
     NpcSetGroup ("eluuny", "Elidral", "#1", "townspeople_eluuny")
     NpcSetGroup ("eluuny", "Elumor", "#1", "townspeople_eluuny")
     NpcSetGroup ("eluuny", "Yveira", "#1", "townspeople_eluuny")
}

alarm SelfDefence
{
     GroupMemberAttacked "townspeople_eluuny", attack
}
```

An important feature of location scripts is the ability to set up alarms. Alarms can be raised when certain conditions are met – player performs a particular action and the game responds with a predefined reaction. Currently, only one type of alarm is supported: when a member of a previously defined group is attacked, the other members will try to kill the attacker. You can observe this in the above script example.

Writing item scripts

The item script is invoked when the player performs some action on an item. For instance, when the player reads a book, the `OnRead` function is called in the script (currently it is the only function supported in the item script). The script file must be present in the Data/Scripts/Items directory. If it does not exist, no further action is performed by the game engine. Example script contents:

```
function OnRead (Type)
{
     if (Type == 181)
     {
```

```
            ReportRead = 1
            MsgBox ("STR_STORY_REPORT")
        }
        else if (Type == 226)
        {
            TextBox ("STR_DIARY")
        }
        else
        {
            return 1
        }

    return 0
}
```

A typical action is to present a message to the player. `MsgBox` is suitable for short messages, while `TextBox` contains scroll bars and thus is able to display a longer piece of texts.

It is expected that you return 0 if the item of the given type is readable. Any other return value (e.g. 1) results in a default message box saying that there was nothing interesting to read (or something similar).

Starting item sets

At character creation, player is able to choose among several item sets that the character will have in their inventory at the beginning of the game. The sets can be defined in the scripts. Two scripts are involved here: the OnLoad script and the global declarations script (**_global.u**).

The first step (in the global declarations file) is to define the item sets. They look just like good old item definitions that can be found in most of dialog scripts. The only difference is that you cannot freely choose their names. They must have the form of "SetX_ItemY", where X is a number of set (from 1 up to the number of sets defined) and Y is an index of item in the set (from 1 up to 3).

```
item "Set1_Item1", "short sword", 1, 0, 0
item "Set1_Item2", "gold piece, 100, 0, 0
item "Set1_Item3", "potion of healing", 3, 0, 0
item "Set2_Item1", "long knife", 1, 0, 0
item "Set2_Item2", "arrow", 100, 0, 0
item "Set2_Item3", "potion of healing", 3, 0, 0
```

Having the sets defined, you now need to handle them in the OnLoad script. You can declare a function named `OnChooseItemSet` that is called just after the character has been created by the player. Its only argument contains an index of the current item set:

```
function OnChooseItemSet (ChosenItemSet)
{
    if (ChosenItemSet == 1)
    {
        AddInvItemByAlias ("Set1_Item1")
        AddInvItemByAlias ("Set1_Item2")
        AddInvItemByAlias ("Set1_Item3")
```

```
    }
    else if (ChosenItemSet == 2)
    {
        AddInvItemByAlias ("Set2_Item1")
        AddInvItemByAlias ("Set2_Item2")
        AddInvItemByAlias ("Set2_Item3")
    }
}
```

In this example the player chooses from two item sets.

Starting equipment is another story. It depends mainly on the chosen weapon skill. For example if the player chooses swords, he will get one, but if they choose staves, they will be equipped with a wand or staff (and some “magical” clothes, most likely). There's another callback function in the OnLoad script for that purpose:

```
function OnChooseWeaponSkill (SkillID)
{
    if (SkillID == 1)  // swords
    {
     // equip a sword
    }
    else if (SkillID == 2)
    // ...etc., etc...
}
```

Functions

ActivateSpecial function

Activates a special object at specified position.

Script types:
all

Syntax:
`ActivateSpecial (loc_uid, x, y)`

Description:
Activates special object at *(x, y)* in location *loc_uid*. See also `DeactivateSpecial`.

AddCompanion function

Adds a monster or NPC to accompany the hero.

Script types:
all

Syntax:
`AddCompanion (npc_or_monster_name, npc_identity)`

Description:
The parameter `npc_or_monster_name` is a name from monster.ini. The second parameter can be omitted for monsters. For NPCs that do not have more than one identity (which is true for most cases), specify "#1" as `npc_identity`.

AddDragon function

Adds a dragon to the specified location.

Script types:
all

Syntax:
`AddDragon (loc_uid, dragon_name, x, y)`

Description:
The `loc_uid` parameter is an UID of a location in which the dragon is going to be added. Position is given by `x` and `y` parameters.
Note that this is currently the only way to add dragon to the game, as you cannot add dragons in the Location Editor (unlike other monsters).

AddGroup function

Adds a new group to the game.

Script types:
all

Syntax:
`AddGroup (group_name)`

AddInvItem function

Adds an item to hero's inventory.

Script types:
all

Syntax:
`AddInvItem (item_name, count)`

Description:
Adds items to hero's backpack. The items are never put in the current equipment (even if there's no room in the backpack and there are some equipment slots free), so you can specify count.

If there are no free slots in the backpack, an exception is thrown (error message appears and the game exits).

AddItem function

Adds an item into specified location (the item is dropped on the ground).

Script types:
all

Syntax:
`AddItem (loc_uid, item_name, x, y)`

Description:
The new item count equals 1. The `loc_uid` parameter can be an empty string; in that case the item is added in the current location.

AddItemEx function

Adds an item into specified location (the item is dropped on the ground). This variant allows to specify the count.

Script types:
all

Syntax:
`AddItemEx (loc_uid, item_name, count, x, y)`

Description:
The `loc_uid` parameter can be an empty string; in that case the item is added in the current location.

AddInvItemByAlias function

Adds an item defined by the `item` command to hero's inventory.

Script types:
all

Syntax:
`AddInvItemByAlias (item_alias)`

Description:
This function requires an item definition given by the `item` command. It makes it possible to add a set of items chosen during the character creation, but can also have other uses.

If there are no free slots in the backpack, an exception is thrown (error message appears and the game exits).

Example:

```
item "my sword", "short sword", 1, 0, 0

AddItemByAlias ("my sword")
```

AddInvItemEx function

Adds an item to hero's inventory.

Script types:
all

Syntax:
`AddInvItemEx (item_name, count, flags, data, material)`

Description:
This functions behaves much like AddInvItem, but allows extra parameters:

`flags` – item flags, explained below
`data` – additional data, explained below
`material` – from what the item has been made (iron, mithril etc.). Available materials are listed in **Data/mat.XX.ini**

Item type	**Purpose of `data`**
Spell books & scrolls	designates spell index from **Data/spells.XX.ini**
Mould	created item type
Other	not used

Item flags are:

Value	**Flag name**	**Description**
1	flgBSIdentified	The blessed/cursed state has been revealed to player.
2	flgBlessed	The item is blessed (better than normal).
4	flgCursed	The item is cursed (worse than normal).
8	flgIdentified	The item is identified (all its special powers and attributes have been revealed to player, except blessed cursed state).
16	flgBSRandom	The blessed/cursed state of the item is determined randomly.
32	flgPoisoned	The item is poisoned.
64	flgActivated	The item has its primary function activated (e. g. a torch is lit)
128	flgVirtual	The item does not exist (do not use explicitly!)

Some of these flags can be combined. You cannot use `flgBSRandom` along with `flgBlessed` or `flgCursed`. An item cannot be blessed or cursed at time (i. e. you cannot combine `flgBlessed` and `flgCursed`). All other combinations are allowed.

Example:

```
AddInvItemEx ("short sword", 1, 0, 0)
```

AddLocLightEffect function

Sets up a special lighting effect in the specified location.

Script types:
all

Syntax:
`AddLocLightEffect (alpha, r, g, b)`

Description:
The color is made using four components (alpha, R, G, B). The alpha component specifies level of opacity and should not be too low. All of the components should be in range 0-255.

The function should be called in `OnNewSession`.

AddLocMsgEffect function

Sets up a special message in the specified location. It is emitted with a specified (random) interval as a log message.

Script types:
all

Syntax:
`AddLocMsgEffect (loc_uid, time, interval, chance, time_flags, str_tag)`

Description:
The effect is added to location *loc_uid*. Time must be equal to *time* (given in seconds) and divisible by *interval*. *Chance* determines probability of the event (the higher the value, the less probable the event). String with the tag *str_tag* is used. For a list of possible values of *time_flags*, see *AddLocSoundEffect*.

AddLocSoundEffect function

Sets up a special sound effect in the specified location. It is emitted with a specified (random) interval as a log message.

Script types:
all

Syntax:
```
AddLocSoundEffect (loc_uid, time, interval, chance, time_flags,
str_tag)
```

Description:
The effect is added to location *loc_uid*. Time must be equal to *time* (given in seconds) and divisible by *interval*. *Chance* determines probability of the event (the higher the value, the less probable the event). String with the tag *str_tag* is used.

`time_flags:`
1 - day only
2 - night only
3 - night and day

AddMonster function

Adds a new monster.

Script types:
all

Syntax:
```
AddMonster (loc_uid, mon_name, x, y)
```

Description:
Adds a new monster to location *loc_uid* and places it at *(x, y)*. The added monster will never leave the specified location (does not follow the hero when they go through a staircase or other portal type).
Returns ID of the added monster or -1 if the location did not exist. The returned ID can be considered as unique (i. e. you can later use it for comparisons, for example to check whether the monster has been killed).

AddMonsterToGroup function

Adds a new monster and assigns it to a group.

Script types:
all

Syntax:
```
AddMonsterToGroup (loc_uid, mon_name, x, y, grp_name)
```

Description:
Adds a new monster to location *loc_uid*, places it at *(x, y)* and assigns it to the group *grp_name*.

AddObject function

Adds a terrain object into the specified location.

Script types:
all

Syntax:
`AddObject (loc_uid, obj_type, x, y)`

Description:
Adds the object of type `obj_type` to location `loc_uid` and places it at `(x, y)`.

`obj_type` – object type (number) from **objects.XX.ini**.

AddTwoArenaMonsters function

Adds two new monsters that will fight each other on an arena.

Script types:
all

Syntax:
`AddTwoArenaMonsters (x1, y1, x2, y2)`

Description:
Adds two new monsters to location `loc_uid` and places them, respectively, at `(x1, y1)` and at `(x2, y2)`.
The function sets the following dialog data:
1 – monster 1 name (string)
2 – monster 2 name (string)
3 – monster 1 ID (number)
4 – monster 2 ID (number)
The script can refer to the string data using "$1" and "$2". Number data can be accessed using the `GetDialDataInt` function.

AnyAliveMonsters function

Checks whether there are any hostile monsters left in the specified location. Dead monsters do not count.

Script types:
all

Syntax:

```
AnyAliveMonsters (loc_uid)
```

Description:
`loc_uid` is the UID of the visited location you want to examine. The functions returns nonzero (true) if the location has never been visited by hero, even if the location contains no alive monsters.

Battle function

Starts a battle between two groups of monsters/NPCs.

Script types:
all

Syntax:
```
Battle (group_1_name, group_2_name, battle_name)
```

Description:
The `battle_name` parameter is used since then to identify the battle event and you may refer to this name whenever you need to refer to some battle (e. g. to retrieve battle results).

BeginQuest function

Adds a new quest log entry.

Script types:
all

Syntax:
```
BeginQuest (quest_id, npc_name, loc_uid, desc_tag)
```

Description:
The `quest_id` can be anything you want, as long as it is unique. Be careful when calling the matching `EndQuest` – it should use exactly the same `quest_id`.
The `npc_name` parameter should be the name of the NPC that charged hero with the quest.
The `loc_uid` parameter should be the UID of the location where the NPC is or where they can be met when the quest is about to be completed.
The `desc_tag` parameter must appear in the string resource file and identifies a string containing the quest's brief description.

See also:
`EndQuest`

Chance function

Returns a non-zero number with a specified probability.

Script types:
all

Syntax:
`Chance (min, max)`

Description:
There is a probability of `min/max` that the function will return 1. Otherwise it returns 0. The function will also return 0 in each one of the following cases:

- if `min` is greater than `max`
- if `min` is negative
- if `max` is negative.

CreateDungeonSite function

Script types:
The OnLoad Script

Syntax:
`CreateDungeonSite (size_x, size_y, num_levels, loc_file_name, enter_x, enter_y)`

Description:
Creates a site of random dungeons. The site consists of several levels (the number is given by `num_levels`). Each level is `size_x` cells wide and `size_y` cells high. The entrance to the dungeon is placed in the location given by name `loc_file_name` (must contain .map extension), at coordinates `enter_x, enter_y`.

The dungeons are randomly generated every time the hero enters them. It means that only one dungeon level at time is actually remembered. If the hero leaves one random dungeon level and travels to another one, the previous one is lost forever, along with its items left behind, monsters etc.

The dungeon site (as you already know, actually it is just one location) is placed at first "free" position in the current world. "Free" means that at the position `size_x * size_y` cells have undefined default terrain type and are not assigned a location loaded from file. If the current world has not enough "free" cells or all the "free" cells are spread through the world, the world is resized to fit our new dungeon location with specified size.

You must call `CreateDungeonSite` AFTER you have called `SetStartLoc`. Otherwise the game will crash. You should call the `CreateDungeonSite` function only in the OnLoad script.

CreatureAddMiscItem function

Adds an item to all matching creatures' inventory.

Script types:
all

Syntax:
`CreatureAddMiscItem (loc_uid, mon_name, item_name, chance, flags)`

Description:
The function is used to add inventory items for all monsters or NPCs in the location.
The function will affect all matching creatures in the location, unless the chance is too small. The chance that a creature will be ignored is calculated as: `chance`/100.
The `flags` are used for creating the item.
The `loc_uid` parameter specifies the location where the creature is to be found. It can be an empty string, which means current location.

CreatureIsAlive function

Returns nonzero if the specified creature is alive.

Script types:
all

Syntax:
```
CreatureIsAlive (creature_id)
```

Description:
This function is location-unsafe (if you use it, you must ensure that the creature exists in the current location).

CreatureSetAttitude function

Sets attitude of a living creature.

Script types:
all

Syntax:
```
CreatureSetAttitude (loc_uid, x, y, attitude)
```

Description:
This is commonly used in dialog scripts to make a creature attack the player or to calm down.
Possible `attitude` values are: 0 – neutral, 1 – friendly, 2 – hostile.
The `loc_uid` parameter specifies the location where the creature is to be found. It can be an empty string, which means current location.

DeactivateSpecial function

Deactivates a special object at specified position.

Script types:
all

Syntax:
```
DeactivateSpecial (loc_uid, x, y)
```

Description:
Deactivates special object at `(x, y)` in location `loc_uid`. See also `ActivateSpecial`.

DefineTradeEntities function

Defines several random entites for NPC's trade window.

Script types:
Trade profiles

Syntax:
`DefineTradeEntities (entities)`

Description:
The functions allows more flexible setup of trade items. For more information, see 'Writing trade profiles'.

DelayedMsgBox function

Makes a message box pop up after the current dialog is finished.

Script types:
NPC scripts

Syntax:
`DelayedMsgBox (msg_tag)`

Description:
If you use the `MsgBox` function in the NPC script, it will appear on top of the dialog window, which is not quite what is usually desired. Thus the `DelayedMsgBox` has been introduced. It does not pop up until the dialog window has been closed.
The `msg_tag` parameter denotes string identifier (the string must be present in the strings.XX.ini file).
Also see `DelayedMsgBox` for a message box that appears after the NPC dialog has been finished.

See also: `MsgBox, TextBox`

DeleteAllItems function

Deletes all items at the specified position in the specified location.

Script types:
all

Syntax:
`DeleteAllItems (loc_uid, x, y)`

Description:
All items lying on the ground at `(x, y)` are removed.

DeleteAllMonsters function

Deletes all monsters in the specified location.

Script types:
all

Syntax:
`DeleteAllMonsters (loc_uid)`

Description:
Monsters are deleted, not killed (no corpses remain!).

DeleteItem function

Deletes a single item at the specified position in the specified location.

Script types:
all

Syntax:
`DeleteItem (loc_uid, item_name, x, y)`

Description:
Only an item with a matching name and lying on the ground at `(x, y)` is removed.

DeleteMonster function

Deletes a monster in the current location.

Script types:
all

Syntax:
`DeleteMonster (id)`

Description:
Only a monster with a matching ID gets deleted. The ID is usually obtained from dialog data (see the appropriate chapter for more information).

DeleteObject function

Deletes a terrain object at the specified position in the specified location.

Script types:
all

Syntax:
`DeleteObject (loc_uid, x, y)`

Description:
Only the object at `(x, y)` is removed, terrain tile remains the same.

DiffDays function

Returns the difference (in days) between the current time and last visit time for the current location.

Script types:
all

Syntax:
`DiffDays()`

Description:
Returns 0 on error (when last visit time is earlier than the current time). The maximum value that can be returned is 86400 (about 240 years). If the actual difference is more than maximum, the maximum value is returned.

DiffHours function

Returns the difference (in hours) between the current time and last visit time for the current location.

Script types:
all

Syntax:
`DiffHours()`

Description:
Returns 0 on error (when last visit time is earlier than the current time). The maximum value that can be returned is 1036800 (about 240 years). If the actual difference is more than maximum, the maximum value is returned.

DiffMonths function

Returns the difference (in months) between the current time and last visit time for the current location.

Script types:
all

Syntax:
`DiffMonths()`

Description:
Returns 0 on error (when last visit time is earlier than the current time). The maximum value that can be returned is 2880 (about 240 years). If the actual difference is more than maximum, the

maximum value is returned.

DiffSeconds function

Returns the difference (in seconds) between the current time and last visit time for the current location.

Script types:
all

Syntax:
`DiffSeconds()`

Description:
Returns 0 on error (when last visit time is earlier than the current time). The maximum value that can be returned is 3732480000 (about 240 years). If the actual difference is more than maximum, the maximum value is returned.

DiffYears function

Returns the difference (in years) between the current time and last visit time for the current location.

Script types:
all

Syntax:
`DiffYears()`

Description:
Returns 0 on error (when last visit time is earlier than the current time). The maximum value that can be returned is 240. If the actual difference is more than maximum, the maximum value is returned.

EndQuest function

Marks the specified quest log entry as 'completed'.

Script types:
all

Syntax:
`EndQuest (quest_id)`

Description:
The `quest_id` must identify a quest log entry previously added by an appropriate `BeginQuest` call.

See also:
`BeginQuest`

EquipAmmo function

Equips hero with specified type of ammo.

Script types:
all

Syntax:
`EquipAmmo (item_name, item_count)`

Description:
This function works much like `EquipItem`, but it includes an additional parameter, `item_count`.

EquipAmmoBS function

Equips hero with specified type of ammo, allowing to specify blessed/cursed state for the ammo.

Script types:
all

Syntax:
`EquipAmmoBS (item_name, item_count, flags)`

Description:
For possible `flags` values, see `AddInvItemEx`.

EquipItem function

Equips hero with specified item

Script types:
all

Syntax:
`EquipItem (item_name)`

Description:
This function creates a new item and places it in the hero's current equipment. For example:

```
EquipItem ("short sword")
```

...Makes the hero start with a short sword in his hand. If we add another call with a "helm" parameter, our hero receives a helm additionally. Of course it is placed on his head, not in his hands. As you can see, the proper part of equipment is matched automatically. If nothing matches the specified item, an exception is thrown (error message appears and the game exits).

If the item cannot be equipped for some reason (for example, the item is a short sword and the hero has currently his right arm chopped off), the item will be added to the inventory. In turn, if the item cannot be placed in the inventory, it will be dropped on the ground under hero's feet.

The function should not be used to add ammo (like arrows, belts etc.), because it doesn't allow to specify item count. In this case use `EquipAmmo` instead.

EquipItemBS function

Equips hero with specified type of item, allowing to specify blessed/cursed state for the item.

Script types:
all

Syntax:
```
EquipItemBS (item_name, flags)
```

Description:
For possible `flags` values, see `AddInvItemEx`.

GiveItem function

Adds specified item to hero's inventory.

Script types:
all

Syntax:
```
GiveItem (item_alias)
```

Description:
The item with the specified alias must be declared first. If hero cannot carry the specified item for whatever reason (e. g. the item is too heavy or there is no more room in the inventory), the item is dropped on the ground.

HasGroupWonBattle function

Returns nonzero if the specified group has won the specified battle. If the battle hasn't been finished yet, zero is returned.

Script types:
all

Syntax:
```
HasGroupWonBattle (group_name, battle_name)
```

Description:
This function is location-unsafe (if you use it, you must ensure that the location of the battle is the current location).

HasHeroEquippedItem function

Checks whether hero has specified item in the current equipment.

Script types:
all

Syntax:
`HasHeroEquippedItem (item_name)`

Description:
Note that this function wants item name, not alias – only raw item names from items.xx.ini are accepted.

HasHeroItem function

Checks whether hero has specified item.

Script types:
all

Syntax:
`HasHeroItem (item_alias)`

Description:
Checks whether hero has specified item. Quantity does matter (i. e. must be equal or greater than the quantity specified in definition of `item_alias)`.

HasHeroSomeItem function

Checks whether hero has specified item.

Script types:
all

Syntax:
`HasHeroSomeItem (item_alias)`

Description:
Checks whether hero has specified item. Quantity does NOT matter (i. e. hero must have at least one item, even if specified item alias defines more than one item).

HeroGainXP function

Increases Hero's experience points by specified number.

Script types:
all

Syntax:
`HeroGainXP (points)`

Description:
none

HeroGetAppearance function

Retrieves hero's appearance attribute.

Script types:
all

Syntax:
`HeroGetAppearance()`

Description:
none

HeroGetDeityStatus function

Retrieves hero's current relations with the specified deity.

Script types:
all

Syntax:
`HeroGetDeityStatus (deity_id)`

Description:
The function returns a number. The bigger it is, the better relations with the deity hero has.
The `deity_id` parameter can be either 324 (Locenax) or 325 (Erysopixyr). Any other value causes the function to return 0.
Important note: if the hero is not aligned with the specified deity, the function will always return 0!

HeroGetDexterity function

Retrieves hero's dexterity attribute.

Script types:
all

Syntax:
`HeroGetDexterity()`

Description:
none

HeroGetIntelligence function

Retrieves hero's intelligence attribute.

Script types:
all

Syntax:
`HeroGetIntelligence()`

Description:
none

HeroGetSkillLevel function

Retrieves hero's skill level.

Script types:
all

Syntax:
`HeroGetSkillLevel (skill_name, level_count)`

Description:
For more details about `skill_name` see `HeroLearnSkill`.

HeroGetStrength function

Retrieves hero's strength attribute.

Script types:
all

Syntax:
`HeroGetStrength()`

Description:
none

HeroGrantChampionStatus function

Grants the hero the status of a champion (of the specified deity).

Script types:
all

Syntax:
`HeroGrantChampionStatus (deity_id)`

Description:
The `deity_id` parameter can be either 324 (Locenax) or 325 (Erysopixyr).

HeroIsAligned function

Checks if the hero is aligned with the current NPC.

Script types:
NPC scripts

Syntax:
`HeroIsAligned()`

Description:
none

HeroIsAnyLimbMissing function

Returns 1 if any of hero's limbs (an arm or leg) is missing.

Script types:
all

Syntax:
`HeroIsAnyLimbMissing()`

Description:
Ignores permanently lost limbs (returns 0 in that case). Broken limbs do not count as missing (0 returned as well).
This function is intended to be used together with the `cure_limb` special redirection.

HeroGetVitality function

Retrieves hero's vitality attribute.

Script types:
all

Syntax:
`HeroGetVitality()`

Description:
none

HeroLearnRecipe function

Makes hero learn a recipe for making the specified item.

Script types:
all

Syntax:
`HeroLearnRecipe (item_name)`

HeroLearnSkill function

Makes hero learn a new skill or advance its level.

Script types:
all

Syntax:
`HeroLearnSkill (skill_name, level_count)`

Description:
`skill_name` is a name stored in **Data/strings.XX.txt** under identifier like STR_OTHER_SKILL_xxx.
If hero did not have specified skill before, they learn it at level `level_count` (usually this parameter is equal to 1). Otherwise, they advance the skill level by `level_count`. The minimum skill level is 1 and maximum is 100. If skill level is already 100, the function has no effect. If `level_count` plus current skill level exceeds 100, the skill level is advanced to 100.

IdentifyItemType function

Identifies all items of the specified type.

Script types:
all

Syntax:
`IdentifyItemType (item_name)`

Description:
This function makes sense only when used with items that needs identifying for the whole type, like magical scrolls, spellbooks or plants. It reveals the true name (and purpose, most of the times) of an

item type, but not blessed/cursed state of the concrete item.

GetDialDataInt function

Retrieves specified numeric dialog data item.

Script types:
all

Syntax:
`GetDialDataInt (id)`

Description:
See the chapter about dialog data for more information.

IsBattleEnded function

Returns nonzero if the specified battle has been ended (all monsters from one of the fighting groups had perished).

Script types:
all

Syntax:
`IsBattleEnded (battle_name)`

Description:
This function is location-unsafe (if you use it, you must ensure that the location of the battle is the current location).

IsLocFirstVisit function

Returns nonzero if the specified location has not been visited by the hero (until now).

Script types:
all

Syntax:
`IsLocFirstVisit (loc_uid)`

Description:
None.

IsPlayerWoman function

Checks hero's gender.

Script types:
all

Syntax:
`IsPlayerWoman()`

Description:
Returns 0 for male or nonzero for female. The gender is determined by the player at the game start-up.

LoadTradeProfile function

Loads specified trade profile, assigns it to current NPC and sets up common NPC's trading parameters.

Script types:
NPC scripts

Syntax:
```
LoadTradeProfile (sell_mult, buy_mult, trade_skill, min_cash,
max_cash)
```

`sell_mult` - Sell multiplier (e. g. 2.0)
`buy_mult` - Buy multiplier (e. g. 0.5)
`trade_skill` - Trade skill (e. g. 1)
`min_cash` - Minimum of cash (e. g. 100)
`max_cash` - Maximum of cash (e. g. 5000)

If you buy an item, its value is multiplied by parameter 1., giving the price (an amount of money you pay to the merchant). It should be greater than or equal to 1, but in some rare circumstances it can be less than 1.

If you sell an item, its value is multiplied by parameter 2., giving the price (an amount of money you receive for the item). It should be less than 1, giving the merchant an opportunity to earn some money for his work.

The parameter 3. determines how hard it would be to negotiate prices with the merchant. A trade skill of 1 is the lowest possible value and means that the NPC doesn't know a thing about how to trade. He would probably accept any crap for any sum of money you propose. But it is very unusual to encounter a merchant like this; you will much more often deal with experienced traders (trade skill up to 100), who will rather take the last shirt from you than allow anybody to cheat them for a single piece of gold.

The parameters 4 and 5 determine how much money will NPC own every time you visit him. Note that the values are not absolute; if you sell the NPC a very precious artifact, he may have much more than the limit until some time has passed and he has had an opportunity to spend the money :-). And if he pays you all the money he has got, he will have 0 gold pieces (even if the minimum is higher) until you revisit him after some time.

LockDoor function

Locks specified door.

Script types:
all

Syntax:
```
LockDoor (uid, x, y)
```

Description:

`uid` is a UID of a location containing the door.
If the door under (x, y) coordinates is not closed, the function makes it closed. If the object under specified coordinates is inactive, or is not a door, or the specified door is already locked, the function has no effect.
Note that this function does not check for a key, so that the door will become locked even if no key has been assigned to the door. Under such circumstances, the door will be inaccessible.

MsgBox function

Displays a message in a modal window.

Script types:
all

Syntax:
`MsgBox (msg_tag)`

Description:
The `msg_tag` parameter denotes string identifier (the string must be present in the strings.XX.ini file).
Also see `DelayedMsgBox` for a message box that appears after the NPC dialog has been finished.

See also: `DelayedMsgBox, TextBox`

MsgLog function

Displays a new message in the message log.

Script types:
all

Syntax:
`MsgLog (msg_text, msg_importance)`

Description:
The `msg_importance` parameter can be one of the following values:
0 – default (let the fate decide ;-))
1 – background information (sounds, special location texts etc.)
2 – do not use, reserved for internal purposes
3 – object information
4 – information (white text, green square)
5 – important message (yellow text, red square)
6 – critical message (red text, red square)

The default (normal) level is 4.

For an example, see the **Writing location scripts** chapter.

Now function

Returns current game time in hours.

Script types:
all

Syntax:
`Now()`

Description:
Can be used to obtain timestamps of certain events.

See also:
`TimestampDiff`

NpcActivate function

Activates specified NPC.

Script types:
all

Syntax:
`NpcActivate (loc_uid, npc_name, npc_identity)`

Description:
`NpcActivate` activates specified NPC identity. Each NPC can have multiple identities (entries in **Data\monsters.XX.ini**). They may differ slightly (for example, have different pictures), but share the same name. Identity name is specified in the **monsters.XX.ini** file as *description* parameter and usually equals '#1'. If you want to add more identities of the same NPC, you usually would name them '#2', '#3' and so on. For more details, please refer the description of **monsters.XX.ini** file.
`NpcDeactivate` does just the opposite.
The l`oc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).

NpcAddEnemy function

Adds specified NPC to the other NPC's enemy list.

Script types:
all

Syntax:
`NpcAddEnemy (loc_uid, npc_name, npc_identity, enemy_name, enemy_identity)`

Description:

For more info about identities, see `ActivateNpc`.
The `loc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).

NpcAddGuardedCell function

Makes specified NPC guard a cell.

Script types:
all

Syntax:
```
NpcAddGuardedCell (loc_uid, npc_name, npc_identity, x, y)
```

Description:
The `loc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
To guard a bigger area, you must call this function multiple times. Loops may be useful here (even nested loops).

NpcAddInvItem function

Adds an item to NPC's inventory.

Script types:
all

Syntax:
```
NpcAddInvItem (npc_name, npc_identity, item_name, item_count)
```

Description:
The player can retrieve the item after killing the NPC. The `item_name` parameter refers to the name in the **items.XX.ini** file.
Note that neither `npc_name` nor `npc_identity` can be empty (as opposed to most of other NPC functions).
For more info about identities, see `ActivateNpc`.

NpcAddSpell function

Enables this NPC to cast the spell.

Script types:
all

Syntax:
`NpcAddSpell (npc_name, npc_identity, spell_name, spell_level)`

Description:
Enables this NPC to cast the spell at specified level. Name of the spell, given as `spell_name` parameter, comes from the **spells.XX.ini** file. Spell level must be 1-100 and designates spell strength (its meaning is different for each spell, but generally: the higher level, the better spell). Note that neither `npc_name` nor `npc_identity` can be empty (as opposed to most of other NPC functions).
For more info about identities, see `ActivateNpc`.

NpcDeactivate function

Deactivates specified NPC.

Script types:
all

Syntax:
`NpcDeactivate (loc_uid, npc_name, npc_identity)`

Description:
See `NpcActivate` for more details.
The l`oc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).

NpcFollowHero function

Makes the current NPC follow the hero.

Script types:
all

Syntax:
`NpcFollowHero (loc_uid, npc_name, npc_identity)`

Description:
See also `NpcUnfollowHero`.
The l`oc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).

NpcIsAlive function

Checks whether specified NPC is alive or not.

Script types:
all

Syntax:
`NpcIsAlive (loc_uid, npc_name, npc_identity)`

Description:
Returns nonzero value if specified NPC is alive or 0 if he (she) has been killed. The `npc_name` specifies NPC's name (like 'Griswold'), while `npc_identity` is usually "#1", which is default identity for all NPCs. The specified creature does not necessarily have to be an NPC as long as it has an unique name.
The `loc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).
For more details about NPC identities, see `NpcActivate` function.

NpcKill function

Kills specified NPC.

Script types:
all

Syntax:
`NpcKill (loc_uid, npc_name, identity)`

Description:
The function just finds a location using given location UID, then finds a NPC with matching name and identity, then makes the NPC dead.
The `loc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).

NpcLoadScript function

Makes sure that the specified NPC's script has been loaded.

Script types:
all

Syntax:
`NpcLoadScript (loc_uid, npc_name, npc_identity)`

Description:
This function is necessary since when hero enters a location that had not been loaded before, all NPC scripts are not loaded either until they are needed. This happens automatically when the hero attempts to chat with the NPC and on several other occassions, but if the NPC script expects any callbacks (e.g. OnKill), it needs to be loaded manually and it is up to you to do that in the right moment!
The `loc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).

NpcRemoveScheduleTasks

Removes scheduled tasks matching the given task ID.

Script types:
all

Syntax:
```
NpcRemoveScheduleTasks (loc_uid, npc_name, npc_identity, task_id)
```

Description:
The function should not be used until future versions of the game engine.
The `task_id` parameter can be any unique integer number. It can be used to identify the scheduled task.
The `loc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).

NpcScheduleTaskGoToLoc

Schedules a new task – going into another location.

Script types:
all

Syntax:
```
NpcScheduleTaskGoToLoc (loc_uid, npc_name, npc_identity, task_id,
tgt_uid, x, y)
```

Description:
The function should not be used until future versions of the game engine.
The `task_id` parameter can be any unique integer number. It can be used to identify the scheduled task.
The `loc_uid` parameter specifies the location where the NPC is to be found. It can be an empty

string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).

NpcScheduleTaskGoToPos

Schedules a new task – going into another cell within the current location.

Script types:
all

Syntax:
`NpcScheduleTaskGoToPos (loc_uid, npc_name, npc_identity, task_id, hour, x, y)`

Description:
The function should not be used until future versions of the game engine.
The `task_id` parameter can be any unique integer number. It can be used to identify the scheduled task.
The l`oc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).

NpcSetActivity

Sets NPC's current activity.

Script types:
all

Syntax:
`NpcSetActivity (loc_uid, npc_name, npc_identity, activity)`

Description:
The `activity` parameter actually stands for AI profile name. For a list of available values, please refer to the chapter “Monster definitions”.

The l`oc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).

NpcSetAttitude function

Changes NPC's attitude towards the hero.

Script types:
all

Syntax:
`NpcSetAttitude (loc_uid, npc_name, npc_identity, attitude)`

Description:
This is commonly used in dialog scripts to make NPC attack if he/she gets angry while talking with the player or to calm down. Possible `attitude` values are: 0 – neutral, 1 – friendly, 2 – hostile.
If `npc_name` is not specified (empty string), the current hero's interlocutor is assumed, but this behavior works properly only when the function is used in dialog scripts.
The `loc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).

NpcSetGroup function

Finds a NPC named *npc_name* (he or she must already exist somewhere, it is not added automatically with this call) and assigns him/her to the group *grp_name*.

Script types:
all

Syntax:
`NpcSetGroup (loc_uid, npc_name, npc_identity, grp_name)`

Description:
The `loc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).
For more info about identities, see `NpcActivate.`

NpcStartChat function

Opens a dialog window and begins a chat with the specified NPC.

Script types:
all

Syntax:
`NpcStartChat (loc_uid, npc_name, npc_identity)`

Description:
Please note that opening a dialog window means invoking a modal loop, so this should be the last command in the script.
The `loc_uid` parameter specifies the location where the NPC is to be found. It can be an empty

string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).
For more info about identities, see `NpcActivate`.

NpcUnfollowHero function

Makes the current NPC stop following the hero.

Script types:
all

Syntax:
`NpcUnfollowHero (loc_uid, npc_name, npc_identity)`

Description:
The `loc_uid` parameter specifies the location where the NPC is to be found. It can be an empty string, which means current location.
Both `npc_name` and `npc_identity` can be empty strings, which denotes current NPC (i. e. the one that is chatting with the player).
See also `NpcFollowHero`.

RestrictMonsterGeneration function

Restricts monster generation in the current location.

Script types:
Location scripts

Syntax:
`RestrictMonsterGeneration (restriction)`

Description:
It is used when monster generation should occur only at a certain time range (for example: only at night). The function also allows to restrict a total number of monsters.
It is recommended to place a call to this function in location's `OnInit` callback function.

Example:
In this example, monsters are generated only from 20 (or 8 P.M.) to 6 A.M. (that is, at night). At most 3 monsters will be
generated.

```
RestrictMonsterGeneration ("time:20-6,number:3")
```

Reveal function

Reveals the current location

Script types:
all

Syntax:
`Reveal (param)`

Description:
`Reveal` just causes hero to see all cells in the current location. If we don't call `Reveal` after setting up a starting location, hero will see only the cells being currently in his FOV (Field Of View).

The `param` value can be safely ignored. It is reserved for future use. You can just put 0 here.

The function needs a location, so calling `Reveal` before a call to the `SetStartLoc` function will result in a nice crash.

Rnd function

Returns a random integer number.

Script types:
all

Syntax:
`Rnd (min, max)`

Description:
If `min` is greater than `max`, the function returns `max`.

SetStartLoc function

Sets starting location.

Script types:
The OnLoad Script

Syntax:
`SetStartLoc (loc_file_name, hero_x, hero_y)`

Description:
Makes the game start in the location loaded from the `loc_file_name` file. The file should have the '.map' extension, which you must also include in the `loc_file_name`. If `loc_file_name` does not contain '.map', the game engine assumes that it is location UID. You should call `SetStartLoc` immediately AFTER the `LoadWorld` function. If you do not keep this order of calling, the behavior of the game will be undefined. You must not call the `SetStartLoc` function more than once. You should call the `SetStartLoc` function only in the OnLoad script.

It also sets up hero's starting position. Our hero will be initially placed at the (`hero_x, hero_y`) cell of the specified location.

SetupItemData function

Sets specified item's special data.

Script types:
all

Syntax:
`SetupItemData (item_alias, data)`

Description:
None

SetupItemMaterial function

Sets specified item's material.

Script types:
all

Syntax:
`SetupItemMaterial (item_alias, material_name)`

Description:
None

Sleep function

Makes the game sleep for a specified amount of time.

Script types:
all

Syntax:
`Sleep (milliseconds)`

Description:
All input is disabled while the application is sleeping. The screen is redrawn before the sleeping is commited, so that all changes invoked from the script before the call for the `Sleep` function are reflected.

TakeItem function

Removes specified item form hero's inventory (if it is there).

Script types:
all

Syntax:
`TakeItem (item_alias)`

Description:
The item with the specified alias must be declared first. If hero does not have the item (or the item count is not sufficient), nothing happens.

TimestampDiff function

Calculates the difference between two given timestamps.

Script types:
all

Syntax:
`TimestampDiff (time1, time2)`

Description:
Timestamps can be obtained using the `Now` function.

TextBox function

Displays (potentially long) text in a window.

Script types:
all

Syntax:
`TextBox (msg_tag)`

Description:
The `msg_tag` parameter denotes string identifier (the string must be present in the strings.XX.ini file).
See also: `DelayedMsgBox, MsgBox`

TutorialShow function

Displays specified tutorial page.

Script types:
all

Syntax:
`TutorialShow (stage_tag)`

Description:

Although this function can be used in all script types, it is the best to place it in the Handler. See the appropriate section for more information,

TutorialIsStage function

Checks if the specified tutorial stage is the current stage.

Script types:
all

Syntax:
`TutorialIsStage (stage_tag)`

Description:

Although this function can be used in all script types, it is the best to place it in the Handler. See the appropriate section for more information.

ValueOf function

Returns value of the specified variable.

Script types:
all

Syntax:
```
ValueOf (var_name)
```

Description:

Allows to retrieve the value of a variable. The variable is found by name. This is useful when the syntax does not allow to use variable reference – especially in variable assignment:

```
var a = 0
var b = a // wrong – syntax error
var c = ValueOf ("a") // OK
```

Keyword list

The following is an alphabetical list of all reserved words in U. You must not use them as identifiers (variable names, action names or function names). Depending on the context, some of these keywords may be actually allowed as identifiers, but don't rely on this behavior, as in the near future this may be a subject of change.

You should also avoid using predefined function names as identifiers (the complete list can be found in this document).

```
action
alarm
call
companion_attack
companion_follow
companion_wait
cls
cure
cure_limb
decr
default
else
end
extends
for
function
goto
i+
```

```
i-
iadd
icmp
if
iin
iloop
incr
input
iout
ishl
ishr
isub
item
null
option
overridden
print
println
repair
return
super
system
text
trade
var
```

Additionally, you must not use this command name:

```
SetDefaultAction
```

INI Files (Game Object Definitions)

General information

Numerous game parameters are kept in INI files. They all reside in the **Data** directory.
Format of Fame INI files is very similar to Windows INI files, but internal (a lot faster) parser is used to deal with them, carrying several additional possibilities.

Several character encoding types can be used, but UTF-8 without BOM is recommended, as it is widely accepted as a portable encoding type.

Sections

INI files are divided into sections. A file must have at least one section, and there must be at least one definition in each section (otherwise, it is ignored). No definition can be placed without a corresponding section.

Section headers

Each section is designated by a header, which looks like:

```
[Name]
```

Header name must only contain alphanumeric characters. Spaces are not allowed. An underscore isn't allowed either.

Numerical values

A definition should look like:

```
name = value
```

White spaces before and after the '=' sign are optional. You can also put more than one space between tokens, this is correct.

The hexadecimal notation is also allowed. It is especially useful for specifying colors. Hexadecimal values start from the '#' character. Color values usually consist of three hexadecimal octets (R, G, B). Example: #E8ECE8.

Strings

Unlike Windows INI, you should double-quote all string values:

```
name = "value"
```

It is not actually a problem for simple parsers to handle strings without quotes, but it looks much more clearly, especially when there are many long definitions in a file.

Data types

INI parser supports five data types: four integer ones (8-bit, 16-bit, signed/unsigned 32-bit) and string. These are: 'int' (-2147483648 to 2147483647), 'unsigned char' or 'uchar' (0 to 255), 'unsigned short' or 'ushort' (0 to 65535), 'unsigned int' or 'uint' (0 to 4294967295). Values with all bits set to 1 are considered invalid.

Comments

All lines starting with semicolon ';' are treated as comments. If any spaces are followed by ';', they are ignored. You cannot place a comment in the same line as a definition.

```
INVALID DEFINITION:

value = 2342345  ; error

VALID DEFINITION:

; OK
value = 2342345
```

Localization

INI file names conform to the following pattern: name.XX.ini, where XX is a language identifier (e. g. en – English). By default, all actual data are contained in Polish INI files, while English ones contain only text strings.

Current game language can be set through the **Data\game.ini** file. You can set *lang_ini* and *lang_script* parameters there as well (this changes language version of INI file that actual numerical parameters are read from).

Monster definitions

Monster definition file has the following format:

[header]
monsters=N

[monster1]
parameter=value1

[monster2]
parameter=value2

...

[monsterN]
parameter=valueM

All parameters are optional, unless explicitly stated that they are required.

Parameter	Data Type	Description	Default
ai	string	AI profile name. See below.	“AI_ProfileHunter”
brain	string	Brain script file.	“monbase.u”
char	string	Character (in the ASCII mode).	-
color	uint	Color (in the ASCII mode).	(gray)
description	string	Identity string for NPCs. In case of monsters it is ignored.	"New Monster"
dexterity	ushort	Monster dexterity. In the current version ignored.	0
dmg1	ushort	Minimum damage caused by monster.	0
dmg2	ushort	Maximum damage caused by monster.	0
family	ushort	Race (humanoids, undeads etc.). See below.	0
features	string	Special features possessed by the monster. See below.	(empty string)
flags	ubyte	Any combination of creature flags. See below.	0
hp1	ushort	Minimum Hit Points	0
hp2	ushort	Maximum Hit Points	0
name	string	Name (e. g. “goblin”). Required.	-
nameb	string	Name in accusative case. Required.	-
named	string	Name in genitive case. Required.	-

Parameter	Data Type	Description	Default
namem	string	Name in plural form. Required.	-
namemd	string	Name in plural form and genitive case. Required.	-
occurrence	ushort	Defines how frequently monster appears in random-generated locations.	5 = very often
picindex	ushort	Picture index in Data\monsters.bmp.	0
regions	string	Defines regions when the monster occurs. See below.	(empty string)
resnormal	ubyte	Melee resistance (0-100%).	0
resmagic	ubyte	Magic resistance (0-100%).	0
resfire	ubyte	Fire resistance (0-100%).	0
rescold	ubyte	Cold resistance (0-100%).	0
resside	ubyte	Resistance to side (sword-like) weapons (0-100%)	0
resranged	ubyte	Resistance to ranged weapons, including all thrown weapons (0-100%)	0
resaxes	ubyte	Resistance to axes (0-100%)	0
resblunt	ubyte	Resistance to blunt weapons, including hammers, mauls, maces, clubs (0-100%)	0
ressilver	ubyte	Resistance to weapons made from silver (0-100%)	0
settlements	string	Random settlement configuration. See below.	(empty string)
sound	string	Description of a sound made typically by the monster. Used as a warning for the player.	(empty string)
soundm	string	Same as above, plural form.	(empty string)
title	string	Title (displayed when looking at the monster).	(empty string)
weight	ushort	Monster weight.	0
wpncateg	ubyte	Weapon category. See below.	0
xp	ulong	Extra experience points (for killing).	0

Melee resistance means resistance to all types of conventional weapon damage (non-magic). This value overrides *resside, resranged, resaxes* and *resblunt* if they are specified as well.

The *ressilver* parameter only applies when the attacker (typically the hero) is using a silver weapon and it overrides all other resistances.

AI profiles

The *ai* parameter value can be:

Value	Description
AI_ProfileHunter	The monster actively seeks a prey (most likely the hero) and immediately attacks if only it can.
AI_ProfileLazy	The monster does nothing unless attacked.
AI_ProfileRanger	The monster is able to attack from distance. It will attempt to come closer to its enemy, but not too close. Otherwise it acts much like AI_ProfileHunter.
AI_ProfileWanderer	The monster wanders around randomly unless attacked.
AI_ProfileWizard	The monster is able to cast spells. Otherwise it acts much like AI_ProfileRanger.

Families

The *family* parameter value can be:

Value	Description
0	Human
1	Humanoid (e. g. goblin)
2	Undead
3	Spider
4	Animal
5	Golem
6	Plant

Monster *family* determines many things. For example golems and undeads never retreat and do not bleed from their wounds and non-humanoids never makes fire when camping in the wilderness.

Weapon categories

The *wpncateg* parameter can affect combat, e. g. monsters using teeth can bite off hero's leg while those who use fists cannot do that. Acceptable values are:

Category	Description
0	Bare fists or claws
1	Sword
2	Axe
3	Hammer
4	Bow or crossbow
5	Thrown weapon

6	Pole weapon
7	Knife
8	Staff
9	Teeth

Flags

Monster flags are:

Flag name	Value	Description
flgUseMonPic	1	NPC uses picture from monsters.bmp (not npc.bmp, which is default)
flgDragon	2	Monster is a dragon (and is bigger than 1 tile).
flgLoadScript	4	NPC script should be always loaded with the location (by default scripts are loaded only on demand).
flgFemale	8	Monster is a female (by default every monster is a male).
flgSummonable	16	Monster (or NPC) can be summoned by a spell.
flgCommonNPC	32	Only applies to NPCs. Common NPCs can have multiple instances at time (normal NPCs must be unique), but unlike monsters they can chat and trade with the hero. They can also have their own inventory and cast spells.
flgCorpseInedible	64	Only applies to NPC definitions. Most of corpses can be eaten by hero (some of them have special effects when eaten, but this is hardcoded in the C++), but some cannot (e. g. corpse of a golem).
flgCanFly	128	Monster can fly (cross the water tiles without drowning or slowing down)
flgCanSwim	256	Monster can swim (cross the water tiles without drowning)

Features

Monster features can be:

Feature Tag	Feature Type	Description
attack-fire	No value	Monster can breath fire.
curse-equip	No value	Monster can curse hero's equipment.
damage-attacker	No value	Monster automatically damages the attacker.
deflect-spells	No value	Monster can deflect offensive spells.
drain-focus	Single value	Monster can drain hero's focus points.

thief-inv	Single value	Monster can steal hero's items (from inventory).
thief-equip	Single value	Monster can steal hero's items (from current equipment).
teleport	No value	Monster can teleport.
vamp	Single value	Monster is a vampire (needs to drink blood, no other food is suitable).
disease-attack	Single value	Monster can spread brain disease.
aggressive	No value	Monster attacks every creature it sees.
bleeding-attack	Single value	Monster can cause bleeding.
spawn-attacked	Single value	Monster can spawn its clones when attacked.
spawn	Single value	Monster can spawn its clones.
confusion	Single value	Monster can cause confusion.
explode	Single value	Monster can explode instead of a normal attack.
attack-poison	Double value	Monster can poison the attacked creature.
attack-paralysis	Double value	Monster can paralyse the attacked creature.

“No value” means that the very presence of the feature tag is required. “Single value” means that a range of number is additionally required, “Double value” means that two ranges are required.

A “No value” feature can be specified as follows:

```
features = "thief"
```

A “Single value” feature can be specified as follows:

```
features = "drain-focus:3-5"
```

Despite the “single” word, a range must be specified. In the above example, the range is 3-5, meaning that 3 to 5 points of focus may be drain in a single turn.

A “Double value” feature can be specified like this:

```
features = "attack-poison:20-50,1-2"
```

Two or more different features can be specified as follows:

```
features = "drain-focus:3-5;curse-equip;deflect-spells"
```

Regions

Certain monsters may only be encountered in specific areas. The parameter “regions” is used to specify these areas for a monster definition. There are three levels of generality that can be used in this parameter's value:

- high level (either “underground” or “wilderness”)
- medium level (terrain type)
- low level (location UID)

These levels can be conbined, making a general rule with some additional places. For instance, to allow a monster to occur in underground, but also in the “burned” terrain, you need to use:

```
regions = "underground,ter:burned"
```

Terrain names should refer to names used in the terrain.XX.ini file.

The lowest level is used as follows:

```
regions = "ter:burned,uid:firetemple1"
```

Location UID can be obtained from the locations.XX.ini file.

The “underground” keyword allows to specify dungeon flavor. For example, to make a monster generate only in flooded dungeons, you should write:

```
regions = "underground:flooded"
```

Every location can have the *flgExplicitMonsterRegion* flag set, which means that only monsters that refer to the same location by its UID in the Regions parameter can be generated there.

Regions are used:
- when generating a random location
- when generating a monster encounter (if the region conditions are not met, the encounter does not happen)

Settlements

Certain monster types are used to occupy random settlements. These monsters have the “settlements” parameter specified. Using this parameter you can define services provided by inhabitants, animals kept by them, shops and treasures found in chests inside buildings.

Keyword	Meaning	Parameters	
		Name	Description
shops	Enables generating shops.	-	-
service	Enables NPCs providing services.	blacksmith	NPC able to repair items.
		healer	NPC able to cure diseases and broken limbs.
treasure	Specifies which items are found in treasure chests.	food	Food.
		potions	Potions.
		gold	Gold.
		weapons	Weapons, armor and

			ammunition.
		ore	Ore.
		materials	Ore, ingots and rods.
		gems	Gems.
		rocks	Ordinary rocks.
		coal	Coal.
		scrolls	Scrolls.
		misc	Various stuff.
link	Makes this monster type linked to another. See below.	-	-
role	Specifies role of an NPC in a settlement. See below.	-	-
animal	Makes this monster an animal kept in settlements. See below.	-	-

Every monster type should have some “cloned” types. Their share most of their monster.ini data, but have different names, for instance: “goblin” and “goblin shopkeeper”, and different talk script. Every kind of service (shopkeeper, blacksmith, healer) needs a separate “clone” type. These “clones” must be linked to the “original” type. Here is an example of linking for a goblin shopkeeper:

```
settlements = "link:goblin,role:shopkeeper"
```

Note: not every monster with a “link” keyword should define its role. You can just link to some type. This makes the monster an inhabitant of the settlement, but without an apparent role.

Animals should link to a monster type, too, but they use a different keyword. Goblins do not have any animals, so let us see how to link a cow to appropriate monster types (humans, trolls and cyclopes):

```
settlements = "animal:citizen,troll,cyclope"
```

Now some examples of other keywords. Here is a settlement setup for goblins. It specifies two types of treasures (food and potions) and enables shops:

```
settlements = "treasure:food,treasure:potions,shops"
```

This is how to enable some additional services:

```
settlements =
"treasure:gold,shops,service:blacksmith,service:healer"
```

Item definitions

Item definition file has the following format:

[header]
items=N

[item1]
parameter=value1

[item2]
parameter=value2

...

[itemN]
parameter=valueM

All parameters are optional, unless directly stated that they are required.

Parameter	Data Type	Description	Default
ac	ubyte	Armor class. Applies only to armors, helms and shields.	0
bonus	string	List of additional special bonuses, see below.	“”
category	ubyte	Category. Full list below.	18
color	uint	Color (in the ASCII mode).	(gray)
data	uint	Additional data. See remarks below.	0
defmat	ubyte	Default material (random material by default). See remarks below.	1
dmg1	ubyte	Minimal damage caused by item. Applies only to weapons.	0
dmg2	ubyte	Maximum damage.	0
durab	ushort	Durability (how much damage an item can receive before being completely destroyed).	0
flags	ubyte	Flags. See below.	0
indefinite	string	Indefinite article (valid for specific languages only, e. g. English)	“a”
invpicindex	ushort	Picture in inventory.	0
name	string	Item name. Required.	“New Item”
nameb	string	Item name in accusative form. Required.	“New Item”
namem	string	Item name in plural form. Required.	“New Item”
namemd	string	Item name in plural possessive form. Required.	“New Item”

Parameter	Data Type	Description	Default
nutrition	ushort	Nutrition. Applies only to food.	0
occurrence	ubyte	Determines if an item will appear in random-generated locations. 0 means that it will not, any non-zero value means that it will appear.	5 = very often
picindex	ushort	Picture in main game screen.	0
value	uint	Item value in shop. Equal to item's price when trader's attributes are: 1.0 and 1.0. This attribute also determines how frequently an item appears in random-generated locations.	0
weight	ushort	Item weight. Small items such as rings may weigh 0.	0

Categories

Item categories are:

Category number	Description
1	Swords
2	Axes
3	Mauls, hammers & maces
4	Bows & crossbows
5	Thrown weapons
6	Pole weapons
7	Knives & daggers
8	Staves
9	Shields
10	Helms & caps
11	Armor
12	Rings
13	Amulets
14	Books
15	Scrolls
16	Potions
17	Valuables
18	Other
19	Food
20	Materials

Category number	Description
21	Tools
22	Explosives
23	Ammo

Data

Data – the meaning of this parameter depends on item's category. It is explained in the table below:

Item category	Meaning of the 'Data' field
Spell books and scrolls	Spell index (please see the 'Spell definitions' chapter)
Moulds and rods	Type of item that can be created using the mould or rod
Unique items	Index of the unique item

Material

Possible materials are defined in the mat.xx.ini file (where xx is language ID). Normally, 0 means “other material”, which is suitable for most of items. All other materials are metals, which are mostly used to create weapons and armor. 1 means “random material” (actually random metal). 2 is typically iron, 3 is copper etc.

Flags

Item flags are:

Flag name	Value	Description
flgUniqueItem	1	Item is unique (artifact).
flgSmallItem	2	Item can be placed in sack.
flgTwoHanded	4	Item must be held in two hands (if weapon).
flgReadable	8	Item is readable.
flgAltCorpse	16	n/a
flgSacrifExcept	16	Item can be sacrificed even if its *occurrence* parameter equals 0.
flgBow	32	Item can fire arrows.
flgCrossbow	64	Item can fire bolts.
flgBottle	128	Item is an empty bottle, vial or flask (not drinkable).
flgLightSource	256	Item is a source of light (torch, lamp etc.)
flgNoIdentify	512	Item does not need to be identified, although it does have

Flag name	Value	Description
		some special properties (bonus)

To combine multiple flags, just add their values, for instance if you want your item to be an unique bow, set its flags to 33 or, more conveniently, to 1|32.

Bonuses

Special item bonuses are defined using the Universal List Format (ULF). The syntax is:

id1:n0_1-n1_1,m0_1-m1_1;id2:n0_2-n1_2,m0_2-m1_2

The "id" parameter specifies a predefined identifier (see the table below). The value of a bonus is a random number from the range n0 to n1. The parameters m0 and m1 specify additional data (currently not used). The format allows an arbitrary number of bonuses, but currently only definitions with 2 bonuses are supported.

Available ids:

id	Description	Min	Max
str	Strength bonus	1	?
dex	Dexterity bonus	1	?
vit	Vitality bonus	1	?
int	Intelligence bonus	1	?
app	Appearance bonus	1	?
nrm	Melee (normal) damage resistance	1	100
fir	Fire resistance	1	100
col	Cold resistance	1	100
mag	Magic resistance	1	100
poi	Poison resistance	1	100
learn	Boost learning	-	-
focus	Faster focus regain	-	-
easy-aim	Bigger chance to hit in a ranged attack	-	-
farsight	Bigger FOV radius	-	-
freemind	Resistance to brain disease	1	100

Examples:

ULF	Resulting bonus
`int:1-2,0-0`	+1 or +2 to Intelligence

`dex:2-2,0-0;str1-1,0-0`	+2 to Dexterity, +1 to Strength
`poi:5-5,0-0;vit1-2,0-0`	+5% Poison Resistance, +1 or +2 to Vitality
`fir:1-10,0-0`	+1-10% Fire Resistance

Spell definitions

Spell definition file has the following format:

[header]
spells=N

[spell1]
parameter=value1

[spell2]
parameter=value2

...

[spellN]
parameter=valueM

All parameters are optional, unless directly stated that they are required.

Parameter	Data Type	Description	Default
energyamount	ushort	Energy amount used to cast spell (do not confuse with focus cost).	1
focuscostbase	ushort	Focus required to cast spell.	5
flags	ushort	Spell flags (explained below).	0
icon	ushort	Picture index used as quick spell icon.	0
name	string	Spell name. Required.	-
needaim	bool	If set to true (nonzero), spell needs aiming.	0
neutrality	ushort	Spell neutrality (0 – offensive, 1 – neutral, 2 – defensive)	1
piccount	ushort	Total number of spell animation frames.	1
picstart	ushort	Index of first image for spell effect animation.	0

You can change parameters of the existing spells if you want to, but please don't bother adding new spells – they will not work automatically.

Flags

The spell flags are:

Flag	Value	Description

flgSummon	1	The spell is a summon spell. It results in a creation of new monsters that will help the caster fighting their enemies.
flgHeroOnly	2	The spell can only be cast by hero. It is not suitable for NPCs.
flgContact	4	The spell affects only creatures standing at 8 cells surrounding the spellcaster.

Terrain definitions

Terrain definition file has the following format:

[header]
terrains=N

[terrain0]
parameter=value1

[terrain1]
parameter=value2

...

[terrainN-1]
parameter=valueM

All parameters are optional, unless directly stated that they are required. Terrain definitions that have name of "???" (exactly three characters) are ignored.

Parameter	Data Type	Description	Default
apmod	ushort	AP modifier. Not used.	0
automapcolor	uint	32-bit color index on AutoMap in RGBA format. Hex value allowed (see: Numerical Values).	0 (black)
char	ubyte	Character (in the ASCII mode)	.
color	uint	Color (in the ASCII mode).	(gray)
flags	ubyte	Terrain flags. See below.	0
name	string	Terrain name. Required.	-
nameb	string	Terrain name in accusative form. Required.	-
picindex	ushort	Index of first picture (of 3 total).	0
deflocname	string	Default location name (it is used when name is not specified by the user). Required.	-

You can specify the special value #DEAD (or 57005) as *automapcolor*. Such an object is drawn on the auto-map using alpha value of 0 (i.e. it is fully transparent). This is useful for objects that shouldn't be displayed on the auto-map.

Flags

Terrain flags are:

Flag	Value	Description
flgSingleTile	1	The terrain entry has only 1 bitmap tile assigned (usually has 3). Example: dungeon floor.
flgInterior	2	The terrain entry represents interior (weather effects does not affect it)
flgWater	8	The terrain entry represents water (hero can dive into it and swim).

Object definitions

Please note that the word "Object" in this document sometimes refers to terrain objects (e. g. trees, walls, stones) and sometimes to game objects in general (including monsters, spells, items and so on). I know this is very confusing. I tried really hard to coin a better term, but failed miserably. You must live with it somehow.

Objects definition file has the following format:

[header]
objects=N

[terrain0]
parameter=value1

[terrain1]
parameter=value2

...

[terrainN-1]
parameter=valueM

All parameters are optional, unless directly stated that they are required.

Parameter	Data Type	Description	Default
automapcolor	uint	32-bit color index on AutoMap in RGBA format. Hex value allowed (see: Numerical Values).	0
char	ubyte	Character (in the ASCII mode)	#
color	uint	Color (in the ASCII mode).	(gray)
flags	ubyte	Object flags. Detailed list below.	0
group	ushort	Object group. Better not use :-).	0
name	string	Terrain name. Required.	-
nameb	string	Terrain name in accusative form. Required.	-
picindex	ushort	Index of first picture (of 3 total).	0
metadata	string	Additional data. See the table below.	""

Metadata

Currently two values have a meaning for objects:

Name	Value count	Description
tiles	2	Specifies number of tiles that together make a bigger object. Makes sense only for big objects, i.e. those in the group 100.
base	1	Base tile index – first object definition for a big object.

Big objects actually consist of a number of 'sub-objects'. The first 'sub-object' of each big object is called a 'base' object and typically has the *metadata* field containing *tiles*, while all other 'sub-objects' have *metadata* with *base*.

Example: a big tree's first 'sub-object' may appear like this:

```
[Terrain46]
name = "big tree"
group = 102
picindex = 101
metadata = "tiles:2-2"
// ...
```

… while three other 'sub-objects' may look like this:

```
[Terrain47] // 48, 49...
name = "big tree"
group = 102
picindex = 102 // 103, 104...
metadata = "base:46"
// ...
```

Flags

Object flags are:

Flag name	Value	Description
flgPassable	1	Object is passable (hero can stand on it).
flgDestructible	2	Not used.
flgFlammable	4	Not used.
flgWater	8	Object represents water (hero can dive into it and swim).
flgObstructFov	16	Object affects hero's Field of Vision (FOV)
flgHurtsKicker	32	Anyone who kicks this objects receives damage.
flgDoor	64	Object is a door (can be closed and opened)

These flags can be combined (see **Item Definitions**).

Group

Defines object's group used by the Location Editor. There are too many objects that can be placed using the editor, so they cannot be shown all at once. Instead the object panel displays only several

available objects on a single page and additionally uses groups for easier navigation.

Group Number	Group Name	Special Group?
0	Miscelanneous	no
1	Village	no
2	Wilderness	no
3	Desert	no
4	Dungeon	no
5	Burned Grounds	no
6	Town	no
7	Fortifications	no
8	(not used)	no
9	(not used)	no
10	(not used)	no
11	(not used)	no
12	Items	no
13	NPCs	no
14	Monsters	no
15	Terrains	no
16	Portals	no
17	Generators	no
18	Signs	no
19	Doors	no
20	Altars	no
21	Lodes	no
100	Big Objects	yes
200	Water Objects	yes
300	Walls	yes
400	Road Objects	yes

Special group numbers are used, as the name cleverly suggests, in a special way. They can be combined with other (“normal”) group numbers, so that they actually appear in both groups in the editor. For example, a section of a wall that typically appears in village buildings may have group number “301”, meaning that it belongs to the group “Village” (1) and the special group “Walls” (300).

For objects in the group “Big Objects”, see the *metadata* attribute.

Hero definitions

Hero definitions specify gender and appearance of the hero. Player can choose between these definitions at the new game start-up. There are no string parameters in a hero definition, so they are language independent. They are stored in `hero.ini`. The file has the following format:

[header]
heroes=N

[hero0]
parameter=value1

[hero1]
parameter=value2

...

[heroN-1]
parameter=valueM

All parameters are optional, unless directly stated that they are required.

Parameter	Data Type	Description	Default
gender	ushort	Gender flag (0=male, 1=female).	0
pic	ushort	Index of the main (default) picture.	0
picdead	ushort	Index of the death picture (displayed when hero is dead).	0
picfrozen	ushort	Index of the frozen picture (displayed when hero becomes frozen by a spell etc.).	0

Encounter definitions

Encounter definitions specify random monster encounters that may happen to the hero while they are traveling through the wilderness. Encounters only apply to random locations in the wilderness. They have no effect on monsters encountered in the underground locations or any kind of predefined locations.

They are stored in enc.XX`.ini`. The file has the following format:

[header]
encounters=N

[enc0]
parameter=value1

[enc1]
parameter=value2

...

[encN-1]
parameter=valueM

All parameters are optional, unless directly stated that they are required.

Parameter	Data Type	Description	Default
EncounterText	string	A text message that is displayed when the hero triggers the encounter.	0
InternalText	string	An internal text message. It is not displayed in the game, but might be used by development tools (e. g. in the editor) or in the code.	0
DifficultyLevel	float	The encounter may happen only if the difficulty level for it matches the difficulty level of the location (i. e. it's not greater than the location level and not significantly lower than it).	1.0
MetaData	string	Various additional data (see below).	(empty string)
*MonTypeX**	ushort	Identifies monster (or NPC) to be encountered.	0
*MonNumLowX**	ushort	Specifies number of monster instances to be created. The actual number is a random integer from the range (*MonNumLowX, MonNumberHighX)*.	0**
*MonNumberHighX**	ushort	See above.	0**
*ScriptX**	string	Script file to be used if the creature to be encountered is a NPC. Ignored for monsters.	(empty string)

* Where X is a consecutive integer number (1, 2, 3 and so on).
** The default value is 1 if X is greater than 1.

The value of `MetaData` defines the behavior of encountered creatures. Possible values are:

- `neutral` – creatures does not attack the player (unless in self-defence)
- `hostile` – creatures attack the player immediately

To create a battle encounter (i.e. one group of creatures fighting another group), you must place at least one monster type with `MetaData` equal to `attackers` and at least one with `MetaData` equal to `defenders`. These values do not combine with `neutral` or `hostile`.

Location definitions

While locations are actually defined using Location Editor and WorldEdit, some of their attributes are stored elsewhere – in the **locations.xx.ini** file (xx is language code). These attributes are:

- section *locnames* – location names (as displayed in the game)
- section *worldparts* – names of bigger world parts (also known as regions)
- section *predefined* – predefined parts of random-generated locations
- section *map* – world map icons for locations

Names are mapped to locations' UIDs (both for *locnames* and *worldparts*).

Sections *predefined* and *map* contain rectangles – two pairs of coordinates (x and y, width and height). Predefined rectangleis the area that does not get overwritten when the location is being generated – its terrain, objects, items and monsters will remain there. However, the area can be moved to some other coordinates. All the stuff outside the 'predefined' rectangle will be overwritten by some random-generated things.

The rectancles in *map* refer to the worldmap.bmp file, which contains the world map, as well as icons that might be displayed on top of it. In this case, the rectangle defines which part of the bitmap contains an appropriate icon representing the location.