Lua documentation for Illarion scripting v4.3

Martin *

2004/2005/2006

*martin@illarion.org, http://www.illarion.org

Contents

1. General

1.1. Formalism

System variables and variables of structures are accessed by ".".
Functions are called by ":".
If a function has no parameters, one still has to write ().
Lines that start with "?" refer to unclear commands.
Lines that start with "!" refer to suggested commands.
For variables, "*r*:" in front of them means reading access, "*rw*:" means reading and writing access.
Names in `this font` refer to illarion-specific key words.
Names in *`this font`* refer to lua-specific key words.
Names in ⟨*in this format*⟩ are placeholder and can be seen as variables.
Names in normal fixed font refer to a special choice of variables.
Names of functions are designed to be self explaining, therefore there are a lot of undocumented functions around.

Examples:

```
XKoordinate=TargetItem.pos.x;
User:talk(CCharacter.say, "Hallo Welt!");
```

Important note: Lua is case sensitive.

1.2. General introduction

Everytime certain events happen (someone shift-clicks an object, a monster dies, someone looks at an object, ... see the section about "entry points"), a script is started. The name of that script is usually defined in the SQL-database in a seperate row. For example, the table common, which holds information about all items in illarion (volume, weight, ...), has a row called com_script, which holds the name of the script that is linked to each item. If someone shift-clicks an item, the lua-script that is linked to this item in common is executed. This script then consists of several functions, defining what happens in certain cases: the item can be used with another item (shift-clicks), with a character and so on. This means, a general item has the following lua-file

```
      -- item.lua
function UseItem(User, SourceItem, TargetItem, counter, param)
    ...
end

function UseItemWithCharacter(User, SourceItem, Character, counter, param)
    ...
end
```

```
function LookAtItem(User, Item)
    ...
end
...
```

Such a lua-file does not need all possible functions; if an item has no `LookAtItem(-)`function (LookAt=left-click), it simply does nothing (special) when looked at. There are also entry points for magic and NPCs, which can be found in the entry points section again.

1.3. Variable types

- ⟨*User*⟩, ⟨*Originator*⟩, ⟨*Character*⟩, `thisNPC(:)` Character-type variables, see chapter "Characters". (thisNPC is a constant for NPC-scripts!)
- ⟨*SourceItem*⟩, ⟨*TargetItem*⟩: Item-type variables, see chapter "Items".
- ⟨*counter*⟩: Holds the value of the clients counter.
- ⟨*param*⟩: Return value for menus. 0 if no menu item was chosen yet (=invoke menu), not 0 for selected item of a menu.
- ⟨*Pos*⟩, ⟨*ItemPos*⟩, ⟨*TargetItemPos*⟩, ⟨*TargetPos*⟩: Position-type variables, see chapter "Positions"

2. Positions

2.1. Function

posStruct ⟨*position*⟩=`position(`*int* ⟨*x*⟩,*int* ⟨*y*⟩,*int* ⟨*z*⟩`)`

Creates position-structure for the point (x,y,z).

2.2. Variables

rw: *int* ⟨*position*⟩.x
rw: *int* ⟨*position*⟩.y
rw: *int* ⟨*position*⟩.z
Usage: XCoordinate=User.pos.x

3. Characters

3.1. Functions

3.1.1. Text/Speech

void ⟨*character*⟩:`talk(`*int* ⟨*texttype*⟩,*text* "⟨*Text*⟩"`)`

⟨*texttype*⟩ can be `CCharacter.say`, `CCharacter.whisper` or `CCharacter.yell`.
Lets a character say/whisper/yell some ⟨*Text*⟩.
Example: `User:talk(CCharacter.say`, "Hello world!"`)`

void ⟨*character*⟩:`inform(`*text* "⟨*Text*⟩"`)`

Used to be "sendMessage": informs a player with a short ⟨*Text*⟩.
Example: `User:inform("`You are drunk.`")`

void ⟨*character*⟩:`introduce(`*chrStruct* ⟨*character2*⟩`)`

Introduces ⟨*character2*⟩ to ⟨*character*⟩

void ⟨*character*⟩:`move(`*int* ⟨*direction*⟩,*boolean* ⟨*active move*⟩`)`

⟨*character*⟩ makes a step into ⟨*direction*⟩. ⟨*active move*⟩ is true if the move was done actively (normal case) and false otherwise. north=0, northeast=1, east=2, southeast=3, south=4, southwest=5, west=6, northwest=7, up=8, down=9; Caution: Currently, only north, south, west and east work!

text ⟨*character*⟩:`alterMessage(`*text* ⟨*Text*⟩,*int* ⟨*LanguageSkill*⟩`)`

Returns the altered ⟨*Text*⟩ with respect to the ⟨*LanguageSkill*⟩ given.

void ⟨*character*⟩:`changeQualityItem(`*int* ⟨*id*⟩,*int* ⟨*amount*⟩`)`

Alters the quality of one item of the given ⟨*id*⟩ in either the belt, body or inventory.

3.1.2. Skills and Attributes

int ⟨*character*⟩:`getSkill(`*text* "⟨*SkillName*⟩"`)`
void ⟨*character*⟩:`increaseSkill(`*int* ⟨*SkillGroup*⟩,*text* "⟨*SkillName*⟩",*int* ⟨*value*⟩`)`

⟨*SkillGroup*⟩: 1=Language, 2=Craftsmanship, 3=Magic, 4=Other, 5=Fighting, 6=Druid, 7=Priest, 8=Bard

void ⟨*character*⟩:`learn(`*int* ⟨*SkillGroup*⟩,*text* "⟨*SkillName*⟩",*int* ⟨*Step*⟩,*int* ⟨*Opponent*⟩`)`

⟨*SkillGroup*⟩: 1=Language, 2=Craftsmanship, 3=Magic, 4=Other, 5=Fighting, 6=Druid, 7=Priest, 8=Bard
⟨*Step*⟩: Determines learning difficulty. Usually set to 2. 1 for very hard skillgain.
⟨*Opponent*⟩: In case of an opponent to learn from (e.g. during a fight) the opponent's skill. Set it 100 otherwise.

int ⟨*character*⟩:`increaseAttrib(`*text* "⟨*AttribName*⟩",*int* ⟨*value*⟩`)`

Increases the attribute given (see below) and returns the new attribute value. Use ⟨*value*⟩=0 to read the attribute's value. Note that this command also sends an playerupdate to all visible characters around, so if you need to change a characters appearance it is a good idea to use this command to make this change visible. This only holds for attributes that are allowed to be changed, like hitpoints.

void ⟨*character*⟩:`tempChangeAttrib(`*text* "⟨*AttribName*⟩",*int* ⟨*value*⟩, *int* ⟨*time*⟩`)`

Increases the attribute given (see below) by a value of ⟨*value*⟩ for a time ⟨*time*⟩ (given in seconds).

void ⟨*character*⟩:`setAttrib(`*text* "⟨*AttribName*⟩",*int* ⟨*value*⟩`)`

"⟨*AttribName*⟩" can be: "faceto", "racetyp" (see below), "sex", "age", "body_height", "attitude", "luck", "strength", "dexterity", "constitution", "agility", "intelligence", "perception", "willpower", "essence", "foodlevel", "hitpoints", "mana", "poisonvalue". And "sex" can be: 0 (=male), 1 (=female), 2 (=neuter) Note that, if you set racetyp, the effect will not be visible immediately. To make it visible, use `increaseAttrib` with an attribute that can be changed, like hitpoints.

int ⟨*character*⟩:`get_race()`

"racetyp" can be: 0 (=human), 1 (=dwarf), 2 (=halfling), 3 (=elf), 4 (=orc), 5 (=lizardman), 6 (=gnome), 7 (=fairy), 8 (=goblin), 9 (=troll), 10 (=mumie), 11 (=skeleton), 12 (=beholder), 13 (=cloud), 14 (=healer), 15 (=buyer), 16 (=seller), 17 (=insects), 18 (=sheep), 19 (=spider), 20 (=demonskeleton), 21 (=rotworm), 22 (=bigdemon), 23 (=scorpion), 24 (=pig), 25 (=unknown1), 26 (=skull), 27 (=wasp), 28 (=foresttroll), 29 (=shadowskeleton), 30 (=stonegolem), 31 (=mgoblin), 32 (=gnoll), 33 (=dragon), 34 (=mdrow), 35 (=fdrow), 36 (=lesserdemon)

int ⟨*character*⟩:`get_face_to()`

"faceto" can be: 0 (=north), 1 (=northeast), 2 (=east), 3 (=southeast), 4 (=south), 5 (=southwest), 6 (=west), 7 (=northwest)

int ⟨*character*⟩:`get_type()`

returns 0 for player, 1 for monster, 2 for NPC

void ⟨*character*⟩:`increasePoisonValue(`⟨*value*⟩`)`
int ⟨*character*⟩:`getPoisonValue()`
void ⟨*character*⟩:`setPoisonValue(`*int* ⟨*value*⟩`)`
int ⟨*character*⟩:`getMentalCapacity()`
void ⟨*character*⟩:`setMentalCapacity(`*int* ⟨*value*⟩`)`
void ⟨*character*⟩:`increaseMentalCapacity(`*int* ⟨*value*⟩`)`
int ⟨*character*⟩:`getMagicType()`

returns MagicType

void ⟨*character*⟩:`setMagicType(`*int* ⟨*MagicType*⟩`)`

MagicType: "mage"=0, "priest"=1, "bard"=2, "druid"=3

int ⟨*character*⟩:`getMagicFlags(`*int* ⟨*MagicType*⟩`)`
void ⟨*character*⟩:`teachMagic(`*int* ⟨*MagicType*⟩,*int* ⟨*MagicFlag*⟩`)`
void ⟨*character*⟩:`LTIncreaseHP(`*int* ⟨*value*⟩,*int* ⟨*count*⟩,*int* ⟨*time*⟩`)`

Every ⟨*time*⟩ seconds, the ⟨*value*⟩ is added to the current hitpoints; that happens ⟨*count*⟩ times, meaning: in ⟨*count*⟩*⟨*time*⟩ seconds, a character gains exactly ⟨*value*⟩*⟨*count*⟩ hitpoints.

void ⟨*character*⟩:`LTIncreaseMana(`*int* ⟨*value*⟩,*int* ⟨*count*⟩,*int* ⟨*time*⟩`)`

LT-effects are additive, one character can drink several health-potions for instance.

int ⟨*character*⟩:`getPlayerLanguage()`

Returns players language. 0 for german, 1 for english, 2 for french (unused).

3.1.3. Quest progress

void ⟨*character*⟩:`setQuestProgress(`*int* ⟨*questID*⟩,*int* ⟨*progress*⟩`)`

A questprogress can be set for a specific quest.

int ⟨*character*⟩:`getQuestProgress(`*int* ⟨*questID*⟩`)`

Returns the questprogress for a specific quest.

3.1.4. Item handling

int ⟨*character*⟩:`createItem(`*int* ⟨*itemID*⟩,*int* ⟨*count*⟩,*int* ⟨*quality*⟩, *int* ⟨*data*⟩`)`

Item is created in the belt or backpack of ⟨*character*⟩. If that is not possible, the items will not be created. The function returns an integer that gives the number of items that cannot be created. `world:createItemFromId` might be a good choice in addition.

void ⟨*character*⟩:`createAtPos(`*int* ⟨*Position_body*⟩,*int* ⟨*itemId*⟩,*int* ⟨*count*⟩`)`

Creates an item at a special body position (see below).

void ⟨*character*⟩:`changeQualityAt(`*int* ⟨*Position_body*⟩,*int* ⟨*qly-amount*⟩`)`

Changes the quality by amount at position_body.

void ⟨*character*⟩:`eraseItem(`*int* ⟨*itemID*⟩,*int* ⟨*count*⟩`)`

⟨*count*⟩ item with ⟨*itemID*⟩ (=number!) are erased from the ⟨*characters*⟩ inventory. You have no influence on which items are deleted, you can just determine ID and number.

int ⟨*character*⟩:`countItem(`*int* ⟨*itemID*⟩`)`
int ⟨*character*⟩:`countItemAt(`*text* ⟨*location*⟩,*int* ⟨*itemID*⟩`)`

Counts only at a certain position; ⟨*character*⟩:countItemAt("all",...) is the same as ⟨*character*⟩:`countItem(...)`
⟨*location*⟩ can be "all", "belt", "body", "backpack".

void ⟨*character*⟩:`increaseAtPos(`*int* ⟨*Position_body*⟩,*int* ⟨*count*⟩`)`
void ⟨*character*⟩:`swapAtPos(`*int* ⟨*Position_body*⟩,*int* ⟨*itemID*⟩,*int* ⟨*quality*⟩`)`

Position_body: `BACKPACK=0, HEAD=1, NECK=2, BREAST=3, HANDS=4, LEFT_TOOL=5, RIGHT_TOOL=6, FINGER_LEFT_HAND=7, FINGER_RIGHT_HAND=8, LEGS=9, FEET=10, COAT=11, LAST_WEARABLE=11`
To be combined with ⟨*Item*⟩:`getType()`.
See fig.(5.1).
If quality=0, then the quality remains the same.

scrItem ⟨*character*⟩:`getItemAt(`*int* ⟨*Position_body*⟩`)`

⟨*Position_body*⟩: `CCharacter.backpack=0, CCharacter.head=1, CCharacter.neck=2, CCharacter.breast=3, CCharacter.hands=4, CCharacter.left_tool=5, CCharacter.right_tool=6, CCharacter.finger_left_hand=7, CCharacter.finger_right_hand=8, CCharacter.legs=9, CCharacter.feet=10, CCharacter.coat=11, CCharacter.belt_pos_1=12, CCharacter.belt_pos_2=13, CCharacter.belt_pos_3=14, CCharacter.belt_pos_4=15, CCharacter.belt_pos_5=16, CCharacter.belt_pos_6=17`
This returns a ScriptItemStruct. See fig. (5.1).

conStruct ⟨*character*⟩:`getBackPack()`

Returns a container-item (which is different from scriptitem and commonitem). Containeritems can be used to pick out items which are placed in it. See ⟨*Container*⟩:`takeItemNr(`⟨*itempos*⟩,⟨*count*⟩`)`.

conStruct ⟨*character*⟩:`getDepot(`*int* ⟨*depotId*⟩`)`

Returns a container-item (the depot of that Character). Containeritems can be used to pick out items which are placed in it. See ⟨*Container*⟩:`takeItemNr(`⟨*itempos*⟩,⟨*count*⟩`)`.

boolean , *scrItem* , *conStruct* ⟨*Container*⟩:`viewItemNr(`*int* ⟨*itempos*⟩`)`

Returns three values in that specific order: `bool` ⟨*success*⟩, `structitem` ⟨*item*⟩, `containeritem` ⟨*container*⟩. ⟨*success*⟩ is ***true*** if Lua was able to get the item, ⟨*item*⟩ holds the item at that position number and ⟨*container*⟩ holds the containerstruct in case the item at that position was a container. This can be used together with ⟨*Container*⟩:`takeItemNr(`⟨*itempos*⟩,⟨*count*⟩`)`.

boolean , *scrItem* , *conStruct* ⟨*Container*⟩:`takeItemNr(`*int* ⟨*itempos*⟩,*int* ⟨*count*⟩`)`

Returns three values in that specific order: `bool` ⟨*success*⟩, `structitem` ⟨*item*⟩, `containeritem` ⟨*container*⟩ and deletes this item (⟨*count*⟩ of them). ⟨*success*⟩ is ***true*** if Lua was able to get the item, ⟨*item*⟩ holds the item at that position number and ⟨*container*⟩ holds the containerstruct in case the item at that position was a container.
Example:

```
TheDepot=User:getDepot(1);
for i=0,30 do
    worked,theItem,theContainer=TheDepot:takeItemNr(i,1);
    if (worked==true) then
        if (theContainer==nil) then
            User:inform("This is no container. It's item-ID is "..theItem.id);
        else
            User:inform("This is a container. It's item-ID is "..theItem.id);
        end
    end
end
```

void ⟨*Container*⟩:`changeQualityAt(`*int* ⟨*itempos*⟩,*int* ⟨*amount*⟩`)`

Changes the quality of an item at a given position inside a container. Returns ***true*** if it worked.

boolean ⟨*Container*⟩:`changeQuality(`*int* ⟨*itemid*⟩,*int* ⟨*amount*⟩`)`

Changes the quality of an item with a given item-ID inside a container. Returns ***true*** if it worked.

boolean ⟨*Container*⟩:`insertContainer(`*scrItem* ⟨*item*⟩,*conStruct* ⟨*container*⟩`)`

Critical command. Inserts a container inside a container. Avoid if possible. Fragile. Returns ***true*** if it worked.

void ⟨*Container*⟩:`insertItem(`*scrItem* ⟨*Item*⟩, *boolean* ⟨*merge*⟩`)`

Inserts an item into a container. Collects identical items which are stackable together to a stack if ⟨*merge*⟩ is ***true***. If there already is an item it will probably be overwritten!

void ⟨*Container*⟩:`insertItem(`*scrItem* ⟨*Item*⟩`)`

Inserts an item which is then placed on the last slot in that container.

int void ⟨*Container*⟩:`countItem(`*int* ⟨*itemid*⟩`)`

Counts the number of items in a container of a given ID. It works recursively, which means that if there is a container in that container containing items of that ID, they are counted as well.

int void ⟨*Container*⟩:`eraseItem(`*int* ⟨*itemid*⟩, *int* ⟨*count*⟩`)`

Erases an amount of items of a given ID. Returns an integer (meaning?).

int void ⟨*Container*⟩:`increaseAtPos(`*int* ⟨*pos*⟩, *int* ⟨*value*⟩`)`

Increases the number of items at a given position. Supposedly returns the number of items afterwards.

boolean void ⟨*Container*⟩:`swapAtPos(`*int* ⟨*pos*⟩, *int* ⟨*newid*⟩, *int* ⟨*newquality*⟩`)`

Changes an item to another one with a new ID, returns ***true*** on success.

int void ⟨*Container*⟩:`weight(`*int* ⟨*rekt=0*⟩`)`;

Returns the maximum weight of that container. ⟨*rekt*⟩ MUST be 0 for technical reasons.

int void ⟨*Container*⟩:`Volume(`*int* ⟨*rekt=0*⟩`)`

Just like `weight`, returns the volume of a container.

3.1.5. All the rest

boolean ⟨*character*⟩:`isInRange(`*chrStruct* ⟨*character2*⟩,*int* ⟨*Distance*⟩`)`

Returns true if ⟨*character2*⟩ is within ⟨*Distance*⟩ of ⟨*character*⟩, else false.

int ⟨*character*⟩:`distanceMetric(`*chrStruct* ⟨*character2*⟩`)`

Returns distance.
Very similar to isInRange, but much more flexible. Better use distanceMetric.

int ⟨*character*⟩:`distanceMetricToPosition(`*posStruct* ⟨*Position*⟩`)`

Returns the distance from ⟨*character*⟩ to ⟨*Position*⟩.

boolean ⟨*character*⟩:`isInRangeToPosition(`*posStruct* ⟨*Position*⟩,*int* ⟨*distance*⟩`)`

Returns ***true*** when the ⟨*character*⟩ is within the ⟨*distance*⟩ to ⟨*position*⟩ and ***false*** otherwise.

void ⟨*character*⟩:`warp(`*posStruct* ⟨*Position*⟩`)`

"Position" is a position-structure as described above.

void ⟨*character*⟩:`forceWarp(`*posStruct* ⟨*Position*⟩`)`

"Position" is a position-structure as described above. This command works exactly as warp, but it ignores any non-passable flags on the target position. That means that you can warp onto e.g. water using this command.

void ⟨*character*⟩:`sendMenu(`*menStruct* ⟨*Menu*⟩`)`

⟨*Menu*⟩ is a menu-structure, created by subsequent application of addItem. See other menu commands

void ⟨*character*⟩:`startMusic(`*int* ⟨*Number*⟩`)`

Starts music.

boolean ⟨*character*⟩:`isAdmin()`

Returns ***true*** if that character is admin (GM) and ***false*** otherwise.

void ⟨*character*⟩:`setClippingActive(`*boolean* ⟨*status*⟩`)`

⟨*status*⟩ must be either ***true*** (walking through walls disabled) or ***false***; this enables the character to walk on fields where he usualy can't walk (water, walls, ...). Please use with care: This has to be turned OFF again!

boolean ⟨*character*⟩:`getClippingActive()`

Returns ***true*** or ***false***.

3.2. Variables

r: *text* ⟨*character*⟩.`lastSpokenText`

Returns this characters last spoken line of text

r: *posStruct* ⟨*character*⟩.`pos`

Position-structure

r: *text* ⟨*character*⟩.`name`
r: *int* ⟨*character*⟩.`id`
r: *boolean* ⟨*character*⟩.`attackmode`

true if character currently attacks, *false* otherwise.

rw: *int* ⟨*character*⟩.`activeLanguage`

"common language"=0, "human language"=1, "dwarf language"=2, "elf language"=3, "lizard language"=4, "orc language"=5, "halfling language"=6, "fairy language"=7, "gnome language"=8, "goblin language"=9, "ancient language"=10

rw: ⟨*character*⟩.`movepoints`

3.3. Constants

chrStruct `thisNPC`

Refers to the NPC whose script is being invoked. It is of a Character-Type.

4. Menues

Caution: Menus do not work with magic spells yet. They only work with items.

4.1. Functions

menStruct `MenuStruct()`

Creates an empty menu structure. Example: `MyMenue=MenuStruct()`;

void ⟨*MenuStruct*⟩:`addItem(`*int* ⟨*ItemID*⟩`)`

Adds the item with the ID ⟨*ItemID*⟩ to the Menustruct. Example: `MyMenue:addItem(17);`

void ⟨*character*⟩:`sendMenu(`*menStruct* ⟨*MenuStruct*⟩`)`

Sends the created ⟨*MenuStruct*⟩ to ⟨*character*⟩. Example: User:`sendMenu(`MyMenue`)`;

Advanced Example:

```
...
List={12,34,53,99,111,189,203};    -- Create a list of item-IDs
newMenu=MenuStruct{};              -- Initialize the menu
for key,value in List do           -- Start loop which picks one item of List after the other...
    newMenu:addItem(List[value]);  -- Add an item to the newMenu
end
User:sendMenu(newMenu);            -- Sends the created menu.
...
```

Handling: see "Lists"

5. Items (scriptItem)

There are two kinds of items in lua. This is of the type scriptItem. These types of item-variables are the parameters in the entry point functions (TargetItem etc.). This kind of item variable holds the individual information about the item (position, ...), but not the general ones (volume, weight, ...). It refers to a individual item (stack). You can, however, identify the commonStruct of an individual item, which can be achieved with the `world:getItemStats(`⟨*scriptItem*⟩`)`. Clearly, the other direction is not possible (gaining knowledge about an individual item via a general item.)

5.1. Functions

int ⟨*Item*⟩:`getType()`

Return values: `notdefined=0, showcase1=1, showcase2=2, field=3, inventory=4, belt=5`

Figure 5.1.: Illustration for positions of items. Red: `itempos`, Green: `getType`

5.2. Variables

r: *chrStruct* ⟨*Item*⟩.`owner`

has the type of ⟨*character*⟩.

r: *posStruct* ⟨*Item*⟩.`pos`

has the type of ⟨*position*⟩, this means that the item lies on the floor.

r: *int* ⟨*Item*⟩.`itempos`

Returns the position of an item if it is at a character.

rw: *int* ⟨*Item*⟩.`id`
rw: *int* ⟨*Item*⟩.`wear`
rw: *int* ⟨*Item*⟩.`quality`
rw: *int* ⟨*Item*⟩.`data`

The data value is an arbitrary value that can be used to individualize items.
Usable for magic weapons, key/lock system, ...
The "quality" of an item is a combination of the actual quality (0-9) and the durability (0-99). A quality value of 999 is a highest quality item (9) with highest durability (99). 123 would mean: a low quality item (1) with not very good durability (23). At durability 0, an item breaks.
Beware: When you set ⟨*Item*⟩.`quality`, you need to create a new item instead the old one:

```
TargetItem.quality=TargetItem.quality-200;
world:changeItem(TargetItem);
```

rw: *int* ⟨*Item*⟩.`number`

The number of items on that stack.

6. Items (commonStruct)

There are unfortunately two types of items. This refers to commonStruct-Items. Note that there are important functions for items in the chapter "World". This kind of item variable holds general information about an item (weight, volume, ID,...), not individual ones like, for example, the current position or things like that. It is, so to say, a generalized item.

6.1. Variables

r: *int* ⟨*Item*⟩.`id`
r: *int* ⟨*Item*⟩.`AgeingSpeed`
r: *int* ⟨*Item*⟩.`Weight`
r: *int* ⟨*Item*⟩.`Volume`
r: *int* ⟨*Item*⟩.`ObjectAfterRot`

> These variables are accessible for common struct items and script items (where they refer to the corresponding common struct item!).
> *Usage*:
> `MyItem.id`, `MyItem.AgeingSpeed`, ...

7. Items (weaponstruct, armorstruct)

7.1. Variables(weapon)

r: *int* ⟨*weaponstruct*⟩.`Attack`
r: *int* ⟨*weaponstruct*⟩.`Defence`
r: *int* ⟨*weaponstruct*⟩.`Accuracy`
r: *int* ⟨*weaponstruct*⟩.`Range`
r: *int* ⟨*weaponstruct*⟩.`WeaponType`
r: *int* ⟨*weaponstruct*⟩.`AmmunitionType`
r: *int* ⟨*weaponstruct*⟩.`ActionPoints`
r: *int* ⟨*weaponstruct*⟩.`MagicDisturbance`
r: *int* ⟨*weaponstruct*⟩.`PoisonStrength`

7.2. Variables (armor)

r: *int* ⟨*armorstruct*⟩.`BodyParts`
r: *int* ⟨*armorstruct*⟩.`PunctureArmor`
r: *int* ⟨*armorstruct*⟩.`StrokeArmor`
r: *int* ⟨*armorstruct*⟩.`ThrustArmor`
r: *int* ⟨*armorstruct*⟩.`MagicDisturbance`
r: *int* ⟨*armorstruct*⟩.`Stiffness`

7.3. Variables (natural armor (monsters))

r: *int* ⟨*naturalarmor*⟩.`strokeArmor`
r: *int* ⟨*naturalarmor*⟩.`thrustArmor`
r: *int* ⟨*naturalarmor*⟩.`punctureArmor`

8. World

8.1. Functions

tleStruct `world:getField(`*posStruct* ⟨*position*⟩`)`

⟨*position*⟩ is a position-structure. The function returns a reference to a field.
Example:

```
Field=world:getField(position(22,10,-3));   -- get reference to "Field"
TileID=Field.tile;                          -- Determine the Tile-ID of that field
```

int `world:getTime(`*text* "⟨*time*⟩"`)`

⟨*time*⟩ can be "year", "month", "day", "hour", "minute", "second"

void `world:erase(`*scrItem* ⟨*Item*⟩,*int* ⟨*amount*⟩`)`

Example 1:
`world:erase(`TargetItem,3`)`
erases 3 items on the TargetItem-Stack if possible
Example 2:
`world:erase(`TargetItem,0`)`
erases the whole TargetItem-Stack. (NOTE: Temporarily *DISABLED*!) If there are not enough of the items to erase, this function returns "false" and does not delete anything.

void `world:increase(`*scrItem* ⟨*Item*⟩,*int* ⟨*count*⟩`)`

Increases the item-counter of ⟨*Item*⟩ (⟨*SourceItem*⟩, ⟨*TargetItem*⟩, ...) for ⟨*count*⟩.

void `world:itemInform(`*chrStruct* ⟨*User*⟩,*scrItem* ⟨*Item*⟩,*text* "⟨*Text*⟩"`)`

Useable in `LookAtItem`: Displays text as item name.

void `world:swap(`*scrItem* ⟨*Item*⟩,*int* ⟨*newItemID*⟩,*int* ⟨*quality*⟩`)`

Exchanges ⟨*Item*⟩ (ScriptItem!) with a new one with ⟨*newItemId*⟩ and ⟨*quality*⟩.

scrItem `world:createItemFromId(`*int* ⟨*ItemID*⟩,*int* ⟨*count*⟩,*posStruct* ⟨*position*⟩,*boolean* ⟨*always-flag*⟩,*int* ⟨*quality*⟩,*int* ⟨*data*⟩`)`

where ⟨*position*⟩ is a position-structure and the always-flag is ***true*** (create also when there is already something on that field) or ***false***, depending on how to create the item. It returns a script item sctruct.

void `world:createItemFromItem(`*scrItem* ⟨*Item*⟩,*posStruct* ⟨*Position*⟩,
varalways-flag`)`

where ⟨*Item*⟩ is of the scriptitem-structure. (NOT of the common-structure! Therefore this IS usable with TargetItem!). It creates an identical copy of a scriptitem.

void `world:createMonster(`*int* ⟨*monsterID*⟩,*posStruct* ⟨*position*⟩,*int* ⟨*movepoints*⟩`)`

⟨*monsterID*⟩: 1=Mumie, 2=Insects, 3=scorpion, 4=skeleton, 5=orc, 6=demonskeleton, 7=beholder, 8=demon, 9=rotworm, 10=spider, 11=sheep, 12=pig, 13=troll, 14=skull, 15=wasp, 16=foresttroll, 17=shadowskeleton, 18=stonegolem, 19=goblin, 20=gnoll, 21=dragon, 22=male_drow, 23=female_drow, 24=lDaemon, 25=Daemonenerscheinung, 27=demonskeleton, 30=Lesser Mummy, 31=Mummy, 32=Old Mummy, 33=Insects, 34=Swamp Insects, 35=Sand Scorpion, 36=Cave Scorpion, 37=Orc, 38=Orc Warrior, 39=Orc Berzerker, 40=Troll, 41=Forest Troll, 42=Weak Skeleton, 43=Skeleton, 44=Skeleton Guardian, 45=Skeleton Fighter, 46=Beholder, 47=Demon Spy, 48=Rotworm, 49=Enormous Spider, 50=Posion Spider, 51=Male Drow, 52=Drow Warrior, 53=Drow Sworddancer, 54=Drow Mistress, 55=Thief, 56=Highwayman, 57=Looter, 58=Immortal, 59=Immortals Ghost, 60=Specter, 61=Demonic Skeleton 62=Demonic Voyeur, 101=runeguardmes, 102=runeguardra, 103=runeguardluk, 104=runeguardtah, 105=runeguardsij, 106=runeguardcun, 107=runeguardqwan, 108=runeguardyeg, 109=runeguardlev, 110=runeguardhept, 111=runeguardkah, 112=runeguardsav, 113=runeguardorl, 114=runeguardkel, 115=runeguardpen, 116=runeguardjus

void `world:makeSound(`*int* ⟨*Number*⟩,*posStruct* ⟨*position*⟩`)`

Starts soundeffect. 1=scream, 2=sheep, 3=sword hit, 4=thunder, 5=bang, 6=chopping wood, 7=fire, 8=smithing, 9=water splash, 10=pouring in (bottle), 11=saw, 12=drink, swallow, 13=snaring noise

void `world:gfx(`*int* ⟨*Number*⟩,*posStruct* ⟨*position*⟩`)`

Starts graphicseffect on ⟨*position*⟩.

void `world:changeTile(`*int* ⟨*TileID*⟩,*posStruct* ⟨*position*⟩`)`
comItem `world:getItemStats(`*scrItem* ⟨*Item*⟩`)`

Returns an commonitem-struct from an ⟨*Item*⟩ that is a scriptitem (like TargetItem). Example:

```
myItem=world:getItemStats(TargetItem)
if (myItem.Weight<100) then
    ...
end
```

comItem `world:getItemStatsFromId(`*int* ⟨*ItemID*⟩`)`

Returns an item-struct like `world:getItemStats(`⟨*Item*⟩`)`

scrItem `world:getItemOnField(`*posStruct* ⟨*Position*⟩`)`

Returns a scriptItem on that field.

boolean `world:isItemOnField(`*posStruct* ⟨*Position*⟩`)` *boolean* `world:isCharacterOnField(`*posStruct* ⟨*Position*⟩`)`

Returns ***true*** for a Character standing on that position and ***false*** otherwise.

chrStruct `world:getCharacterOnField(`*posStruct* ⟨*Position*⟩`)`

Returns a character-struct. See chapter "Characters". Example:

```
...
myPosition=position(122,12,3);
if isCharacterOnField(myPosition) then
    myPerson=getCharacterOnField(myPosition);
    myPerson:talk(CCharacter.say,"You found me!");
end
...
```

chrStruct `world:getPlayersInRangeOf(`*posStruct* ⟨*Position*⟩, *int* ⟨*Range*⟩`)`

Returns a list of character-structs who are in the ⟨*Range*⟩ of ⟨*Position*⟩. See chapter "Characters" and lists in lua.

chrStruct `world:getCharactersInRangeOf(`*posStruct* ⟨*Position*⟩, *int* ⟨*Range*⟩`)`
chrStruct `world:getNPCSInRangeOf(`*posStruct* ⟨*Position*⟩, *int* ⟨*Range*⟩`)`
chrStruct `world:getMonstersInRangeOf(`*posStruct* ⟨*Position*⟩, *int* ⟨*Range*⟩`)`
chrStruct `world:getPlayersOnline()`

Returns a list of character-structs of all players online. See chapter "Characters" and lists in lua.

void `world:changeQuality(`*scrItem* ⟨*ScriptItem*⟩,*int* ⟨*amount*⟩`)`

Changes the quality of a scriptitem (TargetItem, ...) for ⟨*amount*⟩.

void `world:changeTile(`*int* ⟨*tileid*⟩,*posStruct* ⟨*position*⟩`)`

Changes the tile on position-struct "position" to tileid. To be combined with the following command.

void `world:sendMapUpdate(`*posStruct* ⟨*position*⟩,*int* ⟨*range*⟩`)`

Send a map update to all clients of characters that stand in range of that position.

text `world:getItemName(`*int* ⟨*Itemid*⟩,*int* ⟨*PlayerLanguage*⟩`)`

Returns string that represents the itemname of the item with this id in playerlanguage according to table "itemnames".

void `world:changeItem(`*scrItem* ⟨*ScriptItem*⟩`)`

Changes a scriptitem against a new one. Handle with care! Example:

```
function UseItem(User,SourceItem,TargetItem,Counter,Param)
      SourceItem.id = 1               -- we change the source Item to a sword
      SourceItem.quality = 699        -- a really good sword.
      SourceItem.wear = 10            -- a sword wich rots in a very long time
      world:changeItem(SourceItem)    -- now the item is changed
end
```

boolean , *wpnStruct* `world:getWeaponStruct(`*int* ⟨*itemID*⟩`)`

Returns two values: bool (true if it is a weapon) and the weaponstruct of the given item (if there is any).
Example:

```
...
foundWp,MyWeapon=world:getWeaponStruct(1);
if (foundWp==true) then
    User:inform("Attack: " .. MyWeapon.Attack .. " def: " .. MyWeapon.Defence);
end
...
```

boolean , *armStruct* `world:getArmorStruct(`*int* ⟨*itemID*⟩`)`

Returns two values: bool (true if it is an armor) and the armorstruct of the given item.

boolean , *natarmStruct* `world:getNaturalArmor(`*int* ⟨*raceID*⟩`)`

Returns two values: bool (true if that race has natural armor) and the naturalarmorstruct of the given race.

8.2. Variables

r,w: *weatherStruct* `weather`

Returns the current weather.

9. Fields

9.1. Functions

In general, these functions will be combined with `world:getField(`*posStruct* ⟨*position*⟩`)` most of the time.

void ⟨*field*⟩`:swapTopItem(`*int* ⟨*newId*⟩, *int* ⟨*quality*⟩`)`

int ⟨*field*⟩`:countItems()`

Returns the number of items that are placed on top of that field.

scrItem ⟨*field*⟩`:getStackItem(`⟨*stackpos*⟩`)`

Returns the item with position ⟨*stackpos*⟩ (0 being the bottom item) within the pile of items on this field. If ⟨*stackpos*⟩ exceeds the number of items on that field, a 0-item is returned (id=0), therefore it is a good idea to check the number of items on that field first.

Changes the top item on a field to newId with quality. 0 to let the quality unchanged.

void ⟨*field*⟩`:changeQualityOfTopItem(`*int* ⟨*amount*⟩`)`

9.2. Variables

r: *int* `tile`

Returns the tile ID of that field. Recommended use with `world:getField(`⟨*position*⟩`)`.

10. It's a kind of magic

10.1. Global variables

`thisSpell`

Refers to the ID of the spell that is currently casted.

10.2. Some words on magic

Casting a spell is done by selecting one or more runes and eventually selecting a target. We assign numbers to these runes like in fig(10.1). Every spell gets a unique spell-ID which is entirely determined

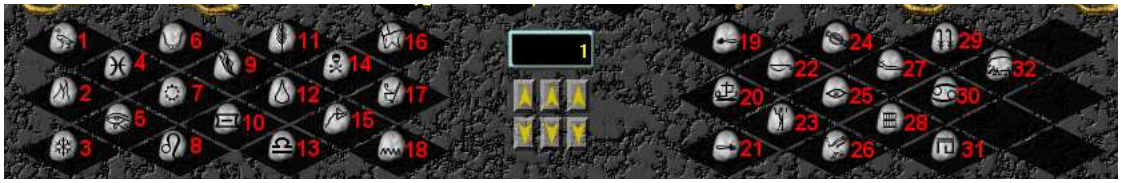

Figure 10.1.: The runes

by the used runes. Suppose we have to use the runes with the numbers $a_1 \dots a_n$ to cast that spell, the spellId can then be calculated by

$$I_{\text{spell}} = \sum_{k=1}^{n} 2^{a_k - 1} = 2^{a_1 - 1} + 2^{a_2 - 1} + \cdots + 2^{a_n - 1}. \tag{10.1}$$

To give a concrete example: Imagine for your spell you have to use runes 2 and 5. The spell Id then is

$$I_{\text{spell}} = \sum_{k=1}^{2} 2^{a_k - 1} = 2^{a_1 - 1} + 2^{a_2 - 1} = 2^{2-1} + 2^{5-1} = 2^1 + 2^4 = 2 + 16 = 17. \tag{10.2}$$

The caption of every spell script should include a brief description of the spell, the rune combination and the SQL insert statement (as comments, of course):
`INSERT INTO spells VALUES(`⟨*spellID*⟩,⟨*magicType*⟩,'⟨*scriptname.lua*⟩')
In our case, that might be: `INSERT INTO spells VALUES(17,0,'m_17_fireball.lua')`

11. Weather

Till now, weather is a global effect. Once you set the weather to a specific value, it's the same everywhere. Eventually there will be "areas" of weather in the future. A weatherStruct is a set of different variables, just like any other struct so far. Altering these variables changes the weather.

11.1. Variables

rw: *int* ⟨*weatherStruct*⟩.`cloud_density`

Varies between 0 (no clouds) and 100 (full clouds).

rw: *int* ⟨*weatherStruct*⟩.`fog_density`
rw: *int* ⟨*weatherStruct*⟩.`wind_dir`
rw: *int* ⟨*weatherStruct*⟩.`gust_strength`
rw: *int* ⟨*weatherStruct*⟩.`percipitation_strength`
rw: *int* ⟨*weatherStruct*⟩.`percipitation_type`
rw: *int* ⟨*weatherStruct*⟩.`thunderstorm`
rw: *int* ⟨*weatherStruct*⟩.`temperature`

11.2. Functions

11.3. Entry point

There is just one entry point for weather scripts: *function* `changeWeather()`

Is invoked everytime the weather should be changed.

12. Long time effects

12.1. Basic idea

The idea of long time effects is the following:

A long time effect is something bound to a character. It can be that you alter some skills or stats at some point (after some action had bin going on, for example after drinking a potion or alike) and undo that change later (after one hour, after 20 seconds, ...). Now, if a character logs out during such an effect is active, all the variables are written to the database and can be read when the character logs on again. Technically, an effect is a collection of different variables, which hold:

- The name and ID of the effect
- A script that defines the effect
- A counter that counts how often an effect had been called already
- A variable that controlls when this effect on this character will be called again
- Several self defined variables that hold all the infos you need

Long time effects are generally handled by a table called `longtimeeffects` with the columns `lte_effectid`, `lte_effectname` and `lte_scriptname`. `lte_effectid` is an arbitrary ID for the effect you are planning, `lte_effectname` is a name which has no technical meaning, you can use it to describe your effect, `lte_scriptname` refers to, as usual, the name of the script that should be invoked.

If you, for example, want to script a potion that adds 10 to a characters perception attribute for the next 10 minutes, you have to do the following: First, create a new entry in the `longtimeeffects`-table, for example with `INSERT INTO longtimeeffects VALUES(15,'mypotion','mypotion.lua')`. The next thing to do is to write a script defining the potions behaviour: It has to create a new effect using the `CLongTimeEffect`-command and then assign this newly created effect to the character with the `addEffect`-command. By doing that, the function `addEffect` (notice the difference between the function and the comman!) in the script assigned to that long time effect (ID) is called (only one time, at the start of the effect!).

There you can, for example, set a variable that determines the value which modifies the perception attribute and also lowers it. After the time given in the `addEffect`-command, the function `callEffect` will be called. If you want the function `callEffect` to be called again, it has to set the value `.nextCall` to the time when you want the function to be called again and return `true`. At the last call, (in this simple case probably the first), it is advisable to set the changed attribute to the correct value again.

However, if the character logs out, the temporary attibute change would have no effect at all because it is not written to the database. Therefore, there is the function `loadEffect`: It is called as soon as a character logs in. At this point, you can read the variable which holds the attribute change with `effect:find` and change it back to the altered value.

12.2. Functions

void ⟨*effect*⟩:`addValue(`*text* ⟨*name*⟩,*int* ⟨*value*⟩`)`

⟨*name*⟩ is an arbitrary name for a variable that can be introduced and filled with ⟨*value*⟩ and is added to that effect. It can later (at one of the following calls, for example) be read or changed again.

void ⟨*effect*⟩:`removeValue(`*text* ⟨*name*⟩`)`

⟨*name*⟩ is the name of a value that will be removed from that effect.

boolean , *int* ⟨*effect*⟩:`findValue(`*text* ⟨*name*⟩`)`

This function returns `true` if a value ⟨*name*⟩ is found plus its value and `false` otherwise. Note that these are two values!

void ⟨*Character*⟩.`effects:addEffect(CLongTimeEffect(`⟨*effect-ID*⟩,*boolean* ⟨*call addEffect*⟩`))`

This function adds the effect ⟨*effect-ID*⟩ to a character. If ⟨*call addEffect*⟩ is `true`, the function `addEffect` will be called.

boolean ⟨*Character*⟩.⟨*effect*⟩:`removeEffect(`⟨*effect-ID*⟩`)`

This function removes the effect ⟨*effect-ID*⟩ from a character. It returns a boolean which indicates whether that worked or not.

12.3. Variables

r: *int* ⟨*effect*⟩.`effectId`
r: *int* ⟨*effect*⟩.`effectName`
r,w: *int* ⟨*effect*⟩.`nextCalled`
r: *int* ⟨*effect*⟩.`lastCalled`
r: *int* ⟨*effect*⟩.`numberCalled`

12.4. Entry points for longtime effects

Inside that script which was invoked, there are several possible entry points that can be called:
function `callEffect(`⟨*Effect*⟩, ⟨*Character*⟩`)`

MUST either return `true` if the effect should be called again or `false` if not! ⟨*Effect*⟩.`nextCalled` has to be set. It will be lowered by 1 every $\frac{1}{10^{\text{th}}}$ second and `callEffect` will be called as soon as it reaches 0.

function `addEffect(`⟨*Effect*⟩, ⟨*Character*⟩`)`

Is invoked when an effect is newly created.

function `removeEffect(`⟨*Effect*⟩, ⟨*Character*⟩`)`

Is invoked after an effect ended (by having `callEffect` return `false`.

function `doubleEffect(`⟨*Effect*⟩, ⟨*Character*⟩`)`

Is invoked when an effect is added to a character that already has that effect. Note that a character can hold just one effect of one type at a time!

function `loadEffect(`⟨*Effect*⟩, ⟨*Character*⟩`)`

Is invoked when a player character logs into the game. It should be used to set temporary stats changes and so on, which can be stored in effect-variables, using `findValue` and so on.

12.5. Adding long time effects to characters

12.6. Example

Imagine the following situation: You drink a potion of a fluid and after that, you get "drunk", that means that your perception and agility are lowered and you sometimes make uncontrolled steps for the next 4 minutes. The first thing to do is to create a table entry in `longtimeeffects` in the following way:

```
 lte_effectid | lte_effectname | lte_scriptname
--------------+----------------+-----------------
          666 | alcohol        | lte_alcohol.lua
```

To start with, we need to script the bottle (`bottle.lua`) which adds the effect 666 (`alcohol`) to the character drinking that bottle.

```
function UseItem(User,SourceItem,TargetItem,Counter,Param,LTstate)

    alcEffect = User.effects:find(666);     -- does effect #666 already exist?

    if (alcEffect == nil) then              -- if that effect is not there...
        alcEffect = CLongTimeEffect(666,10); -- create new effect
        User.effects:addEffect(alcEffect,true);    -- add effect #666
                            -- this calls funct. "addEffect(...)" in the LT-script.
                            -- 1 second until first call, true=invoke addEffect
    end

    alcEffect = User.effects:find(666);     -- does effect #666 exist now?
                                            -- if so, read it into "alcEffect"

    if (alcEffect == nil) then              -- effect not found (security check)
        User:inform("An error occured, inform a developer.");
        return;                             -- exit immediately if not found!
    end

    alcEffect:addValue("alcLevel",10);      -- sets the alcLevel-value to 10.
    alcEffect:addValue("strMod",-5);        -- sets modifier for strength to -5.

end
```

So, this script simply adds `alcEffect` (666) to the `User` of the bottle and adds the value `alcLevel` to this effect and sets it to 10.

The next thing to be done is to define this effect. This is done in the actual long time script we defined in the database before, `lte_alcohol.lua`:

```
function addEffect(myEffect, Character)     -- called only the first time
    Character:inform("You feel a little bit dizzy.");
    found, strMod = myEffect:findValue("strMod");
    if found then                           -- read the str modificator
        Character:increaseAttrib("strength",strMod);
    else                                    -- if modificator is not found
        Character:inform("Error, please inform a developer");
```

```
    end
end

function callEffect(myEffect, Character)    -- is called everytime
    found, alcLevel = myEffect:findValue("alcLevel");
                                            -- get value
    Character:talk(CCharacter.say,"Hick!"); -- Hick!
        -- add some more effects
    if not found then
        Character:inform("Error, please inform a developer!")
        return false;                       -- bug occured!
    else
        if (alcLevel>0) then                -- alcohol still has effect
            myEffect.nextCall=200;          -- next call in 20 s.
            myEffect:addValue("alcLevel",alcLevel-1);
            return true;
        else                                -- alcohol has no more effect
            myEffect:removeValue("alcLevel");
            return false;                   -- return false and delete value
        end
    end
end
```

12.7. Ideas for usage

Long time effects can be used for many effects, here are just some ideas:

- Illness, epidemy, infections and deseases
- Injuries
- Effects of potions (of all kinds)
- Effects of poison
- Punishment
- ...

13. Delayed execution and disturbation

There is a way to have a script being executed after some time. Of course, this could also be done using long time effects, which were described above. However, there is something that long time effects can't detect: If "something" happened between two invokations of an effect. Take, for example, a magician who casts a spell. Assume that there is a delay between casting that spell and having an effect (it needs some time of concentration). What happens if, for example, this mage is disturbed during the concentration phase (because he's under attack or alike)? There is no way to detect that in long time effect scripts.

14. Entry Points

14.1. Items

function `UseItem(`⟨*User*⟩, ⟨*SourceItem*⟩,⟨*TargetItem*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

When one item is used (with another item or alone).

function `UseItemWithCharacter(`⟨*User*⟩, ⟨*SourceItem*⟩, ⟨*Character*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

When an item is used with a character.

function `UseItemWithField(`⟨*User*⟩, ⟨*SourceItem*⟩, ⟨*TargetPos*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

When an item is used with a field (has to be empty=no item lying on the floor!).

function `LookAtItem(`⟨*User*⟩, ⟨*Item*⟩`)`

When someone looks at an item.

function `MoveItemBeforeMove(`⟨*User*⟩, ⟨*SourceItem*⟩, ⟨*TargetItem*⟩`)`

Is invoked when someone tries to move an item before the move is commited. If this function returns *false* the move of the item will not be carried out; that can be used for cursed items. Basically, ⟨*SourceItem*⟩ is the item before it was moved, ⟨*TargetItem*⟩ is the item after it was moved.
`IMPORTANT:` It MUST return either *true* or *false*, otherwise the server crashes! (*return true;*)

function `MoveItemAfterMove(`⟨*User*⟩, ⟨*SourceItem*⟩, ⟨*TargetItem*⟩`)`

Is invoked after someone moved an item. See also `MoveItemBeforeMove(.)`

function `NextCycle()`

Is invoked every 10 seconds for commonitems.

function `CharacterOnField(`⟨*User*⟩`)`

Is invoked if someone steps on that item (which therefore lies on the floor); good for traps and fields. This function requires that the corresponding item has a specialitem-flag in the db-table tilesmodificators

14.2. NPC

For effective usage of NPCs and their scripts please read the section about string handling.

function `nextCycle()`

Is invoked every few server cycles (=approximately constant time intervalls, $\frac{1}{10}$s). *IMPORTANT:* MUST exist in NPC scripts!

function `receiveText(`⟨*TextTyp*⟩, ⟨*Text*⟩, ⟨*Originator*⟩`)`

Is invoked if the NPC hears someone speaking (even himself!).

function `useNPC(`⟨*User*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

Is invoked if the NPC is used (shift-click) by ⟨*User*⟩ without target.

function `useNPCWithCharacter(`⟨*user*⟩, ⟨*targetChar*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`
function `useNPCWithField(`⟨*user*⟩, ⟨*targetChar*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`
function `useNPCWithItem(`⟨*user*⟩, ⟨*targetChar*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

14.3. Magic

function `CastMagic(`⟨*Caster*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

Is invoked when ⟨*Caster*⟩ casts a spell without target.

function `CastMagicOnCharacter(`⟨*Caster*⟩, ⟨*TargetCharacter*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

Is invoked when ⟨*Caster*⟩ casts a spell on another character/monster (⟨*TargetCharacter*⟩).

function `CastMagicOnField(`⟨*Caster*⟩, ⟨*pos*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

Is invoked when ⟨*Caster*⟩ casts a spell on a field at the position ⟨*pos*⟩.

function `CastMagicOnItem(`⟨*Caster*⟩, ⟨*TargetItem*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

Is invoked when a spell is casted on an item.

14.4. Monsters

function `onDeath(`⟨*Monster*⟩`)`

Is invoked as a monster dies.

function `receiveText(`⟨*Monster*⟩, ⟨*TextTyp*⟩, ⟨*Text*⟩, ⟨*Originator*⟩`)`

Is invoked when a monster (⟨*Monster*⟩) receives spoken text.

function `onAttacked(`⟨*Monster*⟩, ⟨*Attacker*⟩`)`

Is invoked when a monster is attacked.

function `onCasted(`⟨*Monster*⟩, ⟨*Caster*⟩`)`

Is invoked when a spell is casted on a monster.

function `useMonster(`⟨*Monster*⟩, ⟨*User*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

Is invoked when a monster is used by ⟨*User*⟩

function `useMonsterWithCharacter(`⟨*Monster*⟩, ⟨*User*⟩, ⟨*TargetChar*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

Is invoked when a monster is used with a character.

function `useMonsterWithField(`⟨*Monster*⟩, ⟨*User*⟩, ⟨*Pos*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

Is invoked when a monster is used with a field.

function `useMonsterWithItem(`⟨*Monster*⟩, ⟨*User*⟩, ⟨*Item*⟩, ⟨*counter*⟩, ⟨*param*⟩`)`

Is invoked when a monster is used with an item.

function `onAttack(`⟨*Monster*⟩,⟨*Enemy*⟩`)`

Is invoked every time when a monster would hit the enemy.

function `enemyOnSight(`⟨*Monster*⟩,⟨*Enemy*⟩`)`

Is invoked every time when a monster sees an enemy. *IMPORTANT*: MUST return true (did something) or false (did nothing)! It is not invoked when the monster stands on a field next to the enemy.

function `enemyNear(`⟨*Monster*⟩,⟨*Enemy*⟩`)`

Is invoked every time when a monster sees an enemy and stands next to it. Works exactly like enemyOnSight, MUST return true or false!

14.5. Fields

function `useTile(`⟨*User*⟩,⟨*Position*⟩,⟨*counter*⟩,⟨*param*⟩`)`

Is invoked when a tile is shift-clicked (used).

function `useTileWithCharacter(`⟨*User*⟩,⟨*Position*⟩,⟨*targetCharacter*⟩,⟨*counter*⟩,⟨*param*⟩`)`

Is invoked when a tile is shift-clicked (used).

function `useTileWithField(`⟨*User*⟩,⟨*Position*⟩,⟨*targetPosition*⟩,⟨*counter*⟩,⟨*param*⟩`)`

Is invoked when a tile is shift-clicked (used).

function `useTileWithItem(`⟨*User*⟩,⟨*Position*⟩,⟨*Item*⟩,⟨*counter*⟩,⟨*param*⟩`)`

Is invoked when a tile is shift-clicked (used).

function `MoveToField(`⟨*User*⟩`)`

Is invoked if a character moves on that triggerfield (entry in "triggerfields" necessary).

function `MoveFromField(`⟨*User*⟩`)`

Is invoked if a character moves away from that triggerfield.

function `PutItemOnField(`⟨*Item*⟩,⟨*User*⟩`)`

Is invoked if an item is put on that triggerfield.

function `TakeItemFromField(`⟨*Item*⟩,⟨*User*⟩`)`

Is invoked if an item is taken away from that triggerfield.

function `ItemRotsOnField(`⟨*oldItem*⟩,⟨*newItem*⟩`)`

Is invoked when ⟨*oldItem*⟩ rots into ⟨*newItem*⟩ on a triggerfield.

14.6. Combat

For combat, there is one main file named standardfighting.lua. It is called everytime a character tries to hit another character. It is important that you cannot reload the DB definitions when this file does not work!

function `BasicFighting(`⟨*Attacker*⟩,⟨*Defender*⟩`)`

Is invoked inside standardfighting.lua if the weapon used has no script assigned to it in table weapons.

function `onAttack(`⟨*Attacker*⟩,⟨*Defender*⟩,⟨*bodyPosition*⟩`)`

Is invoked inside a weapon script if this weapon is used to attack a character at every hit. Don't forget to subtract AP! Currently it is only invoked if the weapon is helt in the right hand, so ⟨*bodyPosition*⟩ will always be the number representing the right hand.

14.7. Player-Characters

There is an entry point that gets called every time a player-character logs into Illarion. There is just one script that is invoked which is named `login.lua`. It is called every time a character logs in.

function `onLogin(`⟨*Character*⟩`)`

This function is invoked inside `login.lua`.

14.8. General

There are ways to invoke arbitrary scripts (functions therein) without player or NPC action.

function ⟨*functionname*⟩()

This script is invoked by having an entry in the database table "`scheduledscripts`" after some given time intervall.
Example:

```
  sc_scriptname   | sc_mincycletime | sc_maxcycletime | sc_functionname
------------------+-----------------+-----------------+----------------
 scheduletest.lua |              14 |              16 | makeeffect
```

This will invoke the function `makeeffect()` in the script `scheduletest.lua` every 14-16 seconds (a random time between 14 and 16 seconds).

15. Lua

15.1. Important commands

For a good summary of the importand commands and how they work look at http://lua-users.org/wiki/TutorialDirectory
Of special interest are: ***for, if, function, while*** and the concept of lists.

15.2. Built in functions

math.random()

Returns a random number between 0 (incl.) and 1 (excl.).

math.random(⟨*Upper*⟩)

Returns a random integer between 1 and ⟨*Upper*⟩ (inclusive).

math.random(⟨*Lower*⟩,⟨*Upper*⟩)

Returns a random integer between ⟨*Lower*⟩ and ⟨*Upper*⟩ (inclusive).

math.abs(⟨*number*⟩)

Returns the absolute value of a number. Example: ***math.abs***-4.2 -¿ 4.2

math.ceil(⟨*number*⟩)

Rounds ⟨*number*⟩ to the next higher integer.

math.floor(⟨*number*⟩)

Rounds ⟨*number*⟩ to the next lower integer.

table.getn(⟨*List*⟩)

Returns the number of entries in a table.

string.find(⟨*text1*⟩,⟨*text2*⟩)

Returns ***nil***, if ⟨*text2*⟩ was not found in ⟨*text1*⟩. If, however, ⟨*text1*⟩ contains ⟨*text2*⟩, it returns the position of the starting character of ⟨*text2*⟩ in ⟨*text1*⟩, the end position of ⟨*text2*⟩ in ⟨*text1*⟩ and, in case one uses so called ***captures***, all the captures found. ***Captures*** are a powerfull concept for stings to analyze them by pattern matching. See the lua wiki.

15.3. Binary operators

`A=B;`

A will get the value of B.

`A==B`

A is compared with B; *true* for A=B else *false*. Used in *if* statements and alike.

`A~=B`

A is compared with B; *false* for A=B else *true* (true if A and B are not equal).

15.4. Lists

Lists are collections of variables. Lists can be created by the following simple procedure: # Start a Lua-list and insert the entries you want (itemIDs etc.) # Run through the list (with a loop) and do what you need (add them to a menue etc.) # Start the process defined by 1. and 2. (send the menue to the player) Creating a list is easy:

`ListA`=⟨*value1*⟩,⟨*value2*⟩,⟨*value3*⟩,...

You can access elements of that table by

`ListA`[⟨*number of element*⟩]

for instance

`ListA[2]`

would be

⟨*value2*⟩

Creating a loop is easy as well:

```
for i = 1,5 do
      ...
end
```

runs through "..." 5 times, the first time with i=1, then i=2, ... to i=5. Combining that with a list would give:

```
ListA={value1,value2,value3,value4,value5}
for i = 1,5 do
    -- do something with ListA[i]
end
```

Example 1: Lets say we want to have a list of items added to a menue.

```
ItemList={45,54,67,81,110,145,215}       -- create list
UserMenu=MenuStruct()                    -- make new menu
for i = 1,7 do                           -- start loop
    UserMenu:addItem ItemList[i]         -- add Item to menu
end
User:sendMenu UserMenu                   -- send menu
```

Example 2: Lets say we want to have a list of items which are only accessible for the player for certain skills. We create a difficulty list.

```
ItemList={45,54,67,81,110,145,215}                        -- create list
DiffList={ 1, 7,45,90, 25, 45, 65}                        -- list of difficulties
UserMenu=MenuStruct()                                     -- make new menu
for i = 1,7 do                                            -- start loop
    if (User:getSkill("smithing")>=DiffList[i]) then      -- if User has enough skill
        UserMenu:addItem(ItemList[i])                     -- add Item to menu
    end
end
User:sendMenu UserMenu                   -- send menu
```

If you are filling a list, you have to take care about the following:
Example 3: Filling a list with entries

```
MyList={};         -- initialize list (IMPORTANT!)
MyList[1]=12;
MyList[2]="Hello";
MyList[3]=56;
...
```

The important part is the first line: Without it, the script would not work. Lets now look to multidimensional lists (tables, for example):
Example 4: Tables

```
MyTable={};
MyTable[1]={};
MyTable[2]={};
MyTable[1][1]=23;
MyTable[1][2]=45;
MyTable[1][3]=34;
MyTable[2][1]="Maoam";
MyTable[2][2]="Hello";
MyTable[2][3]="Hi there!";
```

15.5. Calling functions of other lua files

It is possible to call a function of another (existing) lua file. However, this might cause troubles, as one can only use the whole lua file and it is probable that functions have the same name. This has to be taken care of! To do that, one has to use the ***dofile***("⟨*path*⟩/⟨*filename*⟩") command outside any function. This makes any function in "filename" accessible from within your file. If handled carefully, this may save a lot of work.

Example:
`file_1.lua`:

```
function DoSomething(User,text)
    User:inform(text);
end
```

file_2.lua:

```
dofile("/usr/share/testserver/scripts/file_1.lua")
function TestingDofile(Character)
    DoSomething(Character, "Testing this feature!");
end
```

Note that the testserver scripts are located in `/usr/share/testserver/scripts/` and the realserver scripts in `/usr/share/illarionserver/scripts/`

16. String handling

This is an important topic, as it is relevant for the use of Lua for NPCs (and eventually monsters). The seemably most important function is:
string.find(⟨*text1*⟩, ⟨*text2*⟩)

> Returns a number that indicates the position in text1 of the beginning of the first occurence of text2 in text1 and another number that indicates the last position of the last occurence.

Example:

```
a,b=string.find("Hello world","llo");
   -> a=3, b=5

a,b,c,d=string.find("I buy 20 shoes",".*buy (%d+) (.+)");
   -> a=0, b=14, c="20", d="shoes"
```

- Expressions in brackets "(...)" are returned to the variables. Without them, we would just have a and b.
- "`.`" means: any character, digit, just anything.
- "`*`" means: repetition of the previous, including 0 repetitions; '.*' therefore could mean any string, including an empty one ("").
- "`+`" means nearly the same as '*', except that it has to have at least 1 repetition, therefore the empty string is not included.
- "`%s`" simply means a space (" "). "`a%sb`" therefore means "`a b`".
- "`%d`" means any digit. Together with "`+`" we have "`%d+`", which means: at least one digit, but it can be more.
- "`[Ff]`" would mean: The character must be a "`F`" or a "`f`". "`[Hh][Ee][Ll]+[Oo]`" therefore can be "`hello`" or "`helo`" or "`HeLlo`" or "`heLLLlO`" or...

Therefore the above "`.*buy (%d+) (.+)`" means: Search for a string where you have:

1. any characters or nothing
2. followed by "buy"
3. followed by a space (" ")
4. followed by (at least) one or more digits
5. followed by space
6. followed by one or more characters of any type (could be "shoes", but could also be "!!98(jj" or "hallo" or "9982")

Beware: c and d are both strings, even if they contain a number like "23". If you perform mathematical operations with them (c*2), they behave like numbers, if you compare them (`if c==23`), they behave like strings, meaning that (`c="23"; if c==23 then...`) will NOT work, whereas (`c="23"; if c*1==23 then...`) WILL work, because c*1 is converted into a number.

If a string is not found inside another string, it returns ***nil***.

16.1. File I/O

It is possible to read and write data from/into files. It is important to use files and directories where the scripts are permitted to read and write.

Example:

```
filepoint,errmsg,errno=io.open("/home/martin/scrdata/testing.luadat","r");
thisline=filepoint:read("*line");
User:inform("This line reads as: "..thisline);
filepoint:close();

filepoint,errmsg,errno=io.open("/home/martin/scrdata/testing.luadat","w+");
filepoint:write("User "..User.name.." called that script!");
filepoint:close();
```

For further information see the official lua documentation (`http://www.lua.org`)

17. Examples

17.1. Items

Let us first begin with something simple. Say we want to have a script for a sword with the item-ID 27 (fictional) which, when shift-clicked should simply be deleted. The first thing to do is to create an empty file like "simple_sword.lua" in some text editor (be sure that it uses unix-style end-of-lines!). This file needs a UseItem-function, because the sword should disappear when it is used (shift-clicked). Then we need to write down the command for deleting that item. That's it.

```
-- simple_sword.lua
function UseItem(User, SourceItem, TargetItem, counter, param)
   world:erase(TargetItem,1);
end
```

That will do the job. Now we only need to copy this script to /usr/share/testserver/scripts/ (via svn!) and make an entry in the commons-table of the database into the com_script colum for item 27 which reads simple_sword.lua. Only do a #r inside Illarion's testserver and it works. Let's say that we want to extend our script a little. The character should know that he has deleted something. We add an extra line that informs the player:

```
-- simple_sword.lua
function UseItem(User, SourceItem, TargetItem, counter, param)
   world:erase(TargetItem,1);
   User:inform("You have deleted that damn sword!");
end
```

Copy that file over the old one, do a #r and here we go. As soon as you shift-click the sword, the sword disappears and you get the message "You have deleted that damn sword!". We are still not satisfied with that. Nono. We want to give out some information about that sword, too. It's weight for example. Now, how to do that? This is a little more complex (only a little) because there are two "types" of items that lua knows: the one is the kind of variable like "TargetItem", which does not know anything about it's weight or other properties. The other one knows everything about itself. So we first have to convert TargetItem into such an object. Then we need to get the weight of that. That is done as follows:

```
-- simple_sword.lua
function UseItem(User, SourceItem, TargetItem, counter, param)
   world:erase(TargetItem,1);
   MyItem=world:getItemStats(TargetItem);
   MyWeight=MyItem.Weight;
   User:inform("You have deleted that damn sword which weights "..MyWeight);
end
```

Proceed as before. Let's go on. Next step is: We want to create a new item as soon as the old is destroyed. Let's say the item with the ID 28. Not just one, but, say, 5 of them.

```
-- simple_sword.lua
function UseItem(User, SourceItem, TargetItem, counter, param)
   world:erase(TargetItem,1);
   User:createItem(28,5,333);
   MyItem=world:getItemStats(TargetItem);
   MyWeight=MyItem.Weight;
   User:inform("You have deleted that damn sword which weights ".MyWeight);
end
```

Now, how simple is THAT? Wow. We are still not satisfied. For one reason or another, we do not want to delete that sword in any case. We only want to delete it if the corresponding character does NOT carry a magic key (fictional ID: 30) anywhere on his body. How about that then?

```
-- simple_sword.lua
function UseItem(User, SourceItem, TargetItem, counter, param)
   keys=User:countItem(30);
   if (keys==0) then
      world:erase(TargetItem,1);
      User:inform("You have deleted that damn sword which weights ".MyWeight);
   end
   User:createItem(28,5,333);
   MyItem=world:getItemStats(TargetItem);
   MyWeight=MyItem.Weight;
end
```

This can easily be extended with all the functions and commands listed above. However, let us now turn to more advanced examples. Take, for example, a lockable door. There are several possibilities to lock a door, only limited by the scripters creative mind. The most basic one seems the following: Asume you have two versions of a door: an open one (fictional ID: 50) and a closed one (fictional ID: 51). This door stands on the fictional coordinates (30,30,0). The principle works as follows: closed=locked, open=unlocked, you replace the closed door (50) by the open one (51) if someone uses the correct key (fID: 60) with the closed door. This means that the opening and closing of that door entirely lies in the script of the key (as it is the first object to be shift-clicked!).

```
-- key_door.lua
function UseItem(User, SourceItem, TargetItem, counter, param)
   MyDoor=world:getItemStats(TargetItem);
   if (MyDoor.id==50) then
      MyDoorPosition=TargetItem.pos;
      DesiredPosition=position(30,30,0);
      if (MyDoorPosition=DesiredPosition) then
         world:erase(TargetItem,1);
         world:createItemFromId(51,1,DesiredPosition,true,333)
      end
   elseif (MyDoor.id==51) then
      MyDoorPosition=TargetItem.pos;
      DesiredPosition=position(30,30,0);
      if (MyDoorPosition=DesiredPosition) then
         world:erase(TargetItem,1);
```

```
        world:createItemFromId(50,1,DesiredPosition,true,333)
      end
   end
end
```

This can of course be done in a more elegant way, as the same structure appears twice. However, for learning purposes, this is more obvious: First we check the items ID; if it fits, we check the items position; if that fits, we delete it and create the opened (closed) version instead.

17.2. NPCs

IMPORTANT: NPCs MUST have a `nextCycle()` function, even if it is empty!
Simple NPCs are as simple as simple item scripts. However, they can grow rather large and be arbitrary complex. Lets start with a simple one: He should simply react on "Greetings" or "greetings".

```
function receiveText(texttype, message, originator)
   if string.find(message,"[Gg]reetings") ~= nil then
       thisNPC:talk(CCharacter.say, "Greetings, my friend.");
   end
end
```

Note that we will need string operations intensively. The first one is hidden in "[Gg]reetings", which means that the first letter can be both, a "G" or a "g". If you implement that and test it, the NPC will not react. The reason is rather simple: He does not understand you. In fact, he does not understand any language at all. So we need to increase his language skill, common language preferably. But once he learnd that, he does not need to learn it again, so he only needs to learn it one time. Here comes one thing in quite handy: a script does not forget variables once set; if a variable was never set before, it is nil. So, to check if the variable was set ever before, we need only to check if it is nil.

```
function receiveText(texttype, message, originator)
   if iniVar == nil then
       iniVar=1;
       thisNPC:increaseSkill(1,"common language",100);
   end
   if string.find(message,"[Gg]reetings") ~= nil then
       thisNPC:talk(CCharacter.say, "Greetings, my friend.");
   end
end
```

However, this one will only react on the second string he "hears", because after the first one, he learns common language and doesn't understand anything. Afterwards, he will understand common language. However if we want to add several keywords, we would have an endless and unelegant sequence of if...elseif...elseif...elseif...elseif...end. To avoid that, we can use simple lists where we store the keywords and the reactions and then just loop through them. We do not need to "load" the lists everytime someone talks to our NPC but only once for each server restart, meaning that we could initialize the lists like we increase the language skill of the NPC.

```
function receiveText(texttype, message, originator)
```

```
    if iniVar == nil then
        iniVar=1;
        thisNPC:increaseSkill(1,"common language",100);
        NpcTrig=();
        NpcAnsw=();

        NpcTrig[1]="[Gg]reetings";
        NpcAnsw[1]="Greetings, my friend.";
        NpcTrig[2]="[Hh]ello";
        NpcAnsw[2]="Hello. How are you?";
    end
    for i=1,table.getn(NpcTrig) do
        if string.find(message,NpcTrig[i])~=nil then
            thisNPC:talk(CCharacter.say, NpcAnsw[i]);
        end
    end
end
```

To summarize: We fill the trigger texts and the answers into a list and then search in a loop the received message for a trigger text in that list; if we find one, we let the NPC speak the corresponding answer. We can create very simple dialog. It might be the case, and this is hoped much, that you want to create more complex NPCs than just "question" "answer" things. For example, lets add another thing to this NPC: We want him to do a simple calculation and add together two numbers we tell him. Furthermore we note that, once this NPC found a trigger in a received message, we do not want to search if there is another trigger in the message. We will therefore restructure the for-loop and make a repeat..until-loop instead, which is easier to stop once we found something.

```
function receiveText(texttype, message, originator)
    if iniVar == nil then
        iniVar=1;
        thisNPC:increaseSkill(1,"common language",100);
        NpcTrig={};
        NpcAnsw={};

        NpcTrig[1]="[Gg]reetings";
        NpcAnsw[1]="Greetings, my friend.";
        NpcTrig[2]="[Hh]ello";
        NpcAnsw[2]="Hello. How are you?";
    end
    i=0;
    foundTrig=false;
    repeat
        i=i+1;
        if string.find(message,NpcTrig[i])~=nil then
            thisNPC:talk(CCharacter.say, NpcAnsw[i]);
            foundTrig=true;
        end
    until (i==table.getn(NpcTrig) or foundTrig==true)
    if (foundTrig==false) then
        if(string.find(message,"%d++%d")~=nil then
```

```
            StartsAt,EndsAt,numberOne,numberTwo=string.find(message,"(%d+)+(%d+)";
            thisNPC:talk(CCharacter.say,"This is "..(numberOne+numberTwo));
        end
    end
end
```

18. Common bugs

- Missing *end*, missing (or), script name and db-entry do not match (take care of invisible characters! Try a search in the db with e.g. `SELECT FROM spells WHERE spl_sctiptname='p_28.lua')`
- A "." instead of the seperator ":" or vice versa. ("." is for variables, ":" for functions)
- Missing `()` for functions that don't need parameters.
- Incorrect number of parameters for functions.
- Mispelled function names (use syntax highlighting!).
- Forgot #r in game to reload tables.
- = instead of == or vice versa.
- != instead of =.
- Missing conversion of a string to a number (when reading from a string).
- Using a variable that does not exist in this function (e.g. originator in *function* `nextCycle(...)`
- Beware of endless loops; they freeze the server. Always ensure tha
- Program parts after return statement.
- Forgotten "then" in if-commands, forgotten "end" in inline-if's.

A. Versions

Version 4.3 (30 04 06)
* Added QuestProgress functions
* Added scheduledscripts
* Marked Longtimeeffects as active
* Corrected viewItemNr
Version 4.2 (10 11 05)
* Minor changes and additions (data)

Version 4.1 (23 09 05)
* Added new weapon struct variable
* Added combat functions

Version 4.0.0 (13 08 05)
* Added container commands
* Added variable types
* Removed some minor bugs

Version 3.2.0 (16 07 05)
* Updated dofile
* Added file io
* Updated index
* Changed layout
* Added new graphic

Version 3.1.1 (11 06 05)
* Deleted wrong version of itemInform
* Added index

Version 3.0.1 (11 06 05)
* Deleted wrong version of getItemName

Version 3.0.0 (10 06 05)
* Converted to LaTeXformat
* Deleted unnecessary chapters
* Some additions, correcting mistakes, ...

Version 2.6.2 (02 06 05)
* Minor additions

Version 2.6.0 (29 05 05)
* Added new entry points
* Small corrections
* New commands

Version 2.5.0 (20 04 05)
* Added bugs
* Minor corrections and adaptions

Version 2.4.1 (18 04 05)
* Added further NPC examples
* Corrected minor formating bug
* Minor additions

Version 2.4 (14 04 05)
* Corrected minor errors
* Added new commands
* Added better description of items
* Added starts of NPC tutorial

Version 2.3.1 (06 04 05)
* Minor additions and corrections

Version 2.3 (01 04 05)
* Added a new section for a tutorial
* Minor corrections

Version 2.2.8 (29 03 05)
* Minor corrections
* Minor additions

Version 2.2.7 (27 02 05)
* Converted to WIKI-format
* Some language corrections
* Added some chapters from other versions

Version 2.2.6 (19 11 04)
* Corrected world:makeSound(...)

Version 2.2.5 (15 11 04)
* Corrected world:gfx(...)

Version 2.2.4 (11 11 04)
* Minor additions
* Minor regrouping of skill/attribute-commands

Version 2.2.3 (10 11 04)
* Added chapter "Built in functions"

Version 0.2.2 (09 11 04)
* Minor correction on "world:erase".
* Deleted/Clearified last ?-lines.

Version 0.2.1 (07 11 04)
* Added chapter "Item" and some content

Version 0.2 (07 11 04)
* Added Entry points

Version 0.1.1 (04 11 04)
* Corrected sendMenu-command

Version 0.1 (03 11 04)
* Roughly organized character-commands by topic
* Deleted ⟨*character*⟩:`depot(-)`command
* Changed sendMessage() to inform()
* Added some short descriptions
* German to english translations
* Changed world:erase-command

Index