

# Script Functions

## **AdvMap..... 4**

|                                 |    |
|---------------------------------|----|
| GetGameVar .....                | 4  |
| length .....                    | 4  |
| GetDifficulty .....             | 5  |
| Load .....                      | 5  |
| mod .....                       | 5  |
| print .....                     | 6  |
| Save .....                      | 6  |
| SetGameVar .....                | 6  |
| sleep .....                     | 7  |
| sqrt .....                      | 7  |
| startThread .....               | 8  |
| AddHeroCreatures .....          | 8  |
| AddObjectCreatures .....        | 9  |
| BlockGame .....                 | 9  |
| CalcHeroMoveCost .....          | 10 |
| CanMoveHero .....               | 10 |
| ChangeHeroStat .....            | 10 |
| CreateMonster .....             | 11 |
| DeployReserveHero .....         | 12 |
| EnableAIHeroHiring .....        | 13 |
| EnableHeroAI .....              | 13 |
| IsObjectExists .....            | 14 |
| GenerateMonsters .....          | 14 |
| GetCurrentPlayer .....          | 15 |
| GetDate .....                   | 15 |
| GetHeroCreatures .....          | 16 |
| GetHeroLevel .....              | 16 |
| GetHeroStat .....               | 17 |
| GetObjectCreatures .....        | 17 |
| GetObjectiveProgress .....      | 18 |
| GetObjectiveState .....         | 18 |
| GetObjectOwner .....            | 19 |
| GetObjectsInRegion .....        | 19 |
| GetObjectPosition .....         | 20 |
| GetPlayerHeroes .....           | 20 |
| GetPlayerResource .....         | 21 |
| GetTownBuildingLevel .....      | 21 |
| GetTownBuildingLimitLevel ..... | 23 |
| GetTownBuildingMaxLevel .....   | 24 |
| GetTownHero .....               | 24 |
| GiveArtefact .....              | 25 |
| LevelUpHero .....               | 25 |
| GiveHeroSkill .....             | 25 |
| GiveHeroWarMachine .....        | 26 |
| HasArtefact .....               | 26 |
| HasBorderguardKey .....         | 27 |
| HasHeroSkill .....              | 27 |

|                                |    |
|--------------------------------|----|
| HasHeroWarMachine .....        | 28 |
| IsHeroAlive .....              | 28 |
| IsHeroLootable .....           | 28 |
| IsObjectEnabled.....           | 29 |
| IsObjectInRegion.....          | 29 |
| IsObjectiveVisible .....       | 30 |
| IsObjectVisible .....          | 30 |
| IsRegionBlocked .....          | 31 |
| KnowHeroSpell .....            | 31 |
| Loose .....                    | 31 |
| MarkObjectAsVisited.....       | 32 |
| MessageBox .....               | 32 |
| MoveCamera .....               | 33 |
| MoveHero .....                 | 33 |
| MoveHeroRealTime .....         | 34 |
| GetAllNames .....              | 34 |
| OpenCircleFog .....            | 35 |
| OpenRegionFog.....             | 35 |
| Play2DSound .....              | 35 |
| Play3DSound .....              | 36 |
| PlayObjectAnimation .....      | 36 |
| random.....                    | 37 |
| RazeTown.....                  | 37 |
| RegionToPoint.....             | 38 |
| RemoveArtefact.....            | 38 |
| RemoveHeroCreatures.....       | 39 |
| RemoveHeroWarMachine .....     | 39 |
| RemoveObject .....             | 40 |
| RemoveObjectCreatures .....    | 40 |
| ResetHeroCombatScript .....    | 41 |
| ResetObjectFlashlight.....     | 41 |
| SetAIHeroAttractor .....       | 42 |
| SetAIPlayerAttractor .....     | 42 |
| SetCombatLight.....            | 43 |
| SetHeroCombatScript.....       | 43 |
| SetHeroLootable.....           | 44 |
| SetAmbientLight .....          | 44 |
| SetObjectEnabled .....         | 45 |
| SetObjectiveProgress.....      | 45 |
| SetObjectiveState .....        | 46 |
| SetObjectiveVisible .....      | 47 |
| SetObjectFlashlight .....      | 47 |
| SetObjectOwner .....           | 47 |
| SetObjectPosition .....        | 48 |
| SetPlayerResource .....        | 48 |
| SetPlayerStartResources .....  | 49 |
| SetRegionBlocked .....         | 50 |
| SetTownBuildingLimitLevel..... | 50 |
| SetWarfogBehaviour .....       | 51 |
| ShowFlyingSign .....           | 51 |
| SiegeTown.....                 | 51 |
| StartCombat.....               | 52 |

|                        |    |
|------------------------|----|
| StartCutScene .....    | 53 |
| StartDialogScene ..... | 54 |
| StopPlaySound .....    | 54 |
| TeachHeroSpell .....   | 54 |
| TransformTown .....    | 55 |
| Trigger .....          | 55 |
| UnblockGame .....      | 57 |
| UnreserveHero .....    | 58 |
| Win .....              | 58 |

## **COMBAT..... 58**

|                              |    |
|------------------------------|----|
| Prepare .....                | 59 |
| Start .....                  | 59 |
| IsHuman .....                | 59 |
| IsComputer .....             | 59 |
| SetControlMode .....         | 60 |
| EnableAutoFinish .....       | 60 |
| Finish .....                 | 61 |
| GetAttackerHero .....        | 61 |
| GetAttackerCreatures .....   | 61 |
| GetAttackerWarMachines ..... | 62 |
| GetAttackerWarMachine .....  | 62 |
| GetDefenderHero .....        | 63 |
| GetDefenderCreatures .....   | 63 |
| GetDefenderWarMachines ..... | 63 |
| GetDefenderWarMachine .....  | 63 |
| GetDefenderBuildings .....   | 64 |
| GetDefenderBuilding .....    | 64 |
| IsAttacker .....             | 65 |
| IsDefender .....             | 65 |
| IsHero .....                 | 66 |
| IsCreature .....             | 66 |
| IsWarMachine .....           | 66 |
| IsBuilding .....             | 67 |
| GetHeroName .....            | 67 |
| GetCreatureType .....        | 67 |
| GetCreatureNumber .....      | 68 |
| GetWarMachineType .....      | 68 |
| GetBuildingType .....        | 68 |
| GetUnitPosition .....        | 69 |
| AddCreature .....            | 69 |
| EnableCinematicCamera .....  | 70 |

## **Tutorial..... 70**

|                                |    |
|--------------------------------|----|
| IsTutorialItemEnabled .....    | 70 |
| IsTutorialMessageBoxOpen ..... | 71 |
| TutorialActivateHint .....     | 71 |
| TutorialMessageBox .....       | 71 |
| TutorialSetBlink .....         | 72 |

|                          |                  |
|--------------------------|------------------|
| <b><u>Town</u></b> ..... | <b><u>72</u></b> |
| HeroHired .....          | 72               |
| CreatureHired .....      | 72               |

---



---

## ADVMAP

---



---

### GetGameVar

GetGameVar – return the value of the game’s general variable

#### Syntax

**GetGameVar** ( name );

#### Description

This function returns the value of the game’s general variable with name *name* if there is such, or a null string if there isn’t.

*name* – variable’s name

---

### length

length – determine the length of the array (table)

#### Synopsis

**length** ( array );

#### Description

Let us determine the meaning of ‘array’ as a special case of a talbe, in which the keys are only numbers, and the numeration starts from zero and goes on without missed elements. The *length* function will determine the number of elements in this array.

*array* – the array

---

# GetDifficulty

GetDifficulty – determine the game's difficulty level

## Syntax

```
GetDifficulty(void);
```

## Description

This function returns the game's difficulty level. The following constants are set to identify the difficulty levels: `DIFFICULTY_NORMAL`, `DIFFICULTY_HARD`, `DIFFICULTY_HEROIC`. The constants are sorted by growth, which makes it possible to use constructions like `GetDifficulty() > DIFFICULTY_NORMAL`, etc.

---

# Load

Load – Load the game from the specified file

## Syntax

```
Load(fileName);
```

## Description

This function loads the game from the specified file.

*fileName* is the name of the text file specified in the map's properties (*resources->saveFileNames*) and containing the localized name of the game's save file.

---

# mod

mod – return the remainder from dividing one number to the other.

## Syntax

```
mod(x, y);
```

## Description

This function returns the remainder of the division of the first argument to the second one. Both arguments must be numbers (not necessarily whole). The second argument must be not equal to zero.

---

# print

print – show the textual presentation of the arguments in the console.

## Syntax

```
print(...);
```

## Description

This function shows the textual presentation of the argument(s) in the console.

print works with any type of parameters:

- numbers, strings and nil are presented literally
- functions and userdata objects are substituted with markers like "<function>"
- for the tables, their contents are shown

Example:

```
> print(2, " sdf ", print)
2 sdf <C-function>
```

---

# Save

Save – saves the game into the specified file

## Syntax

```
save(fileName);
```

## Description

This function saves the game into the specified file.

*fileName is the name of the text file specified in the map's properties (resources->saveFileNames) and containing the localized name of the game's save file.*

---

# SetGameVar

SetGameVar – set the value of the game's general variable.

## Syntax

```
SetGameVar(name, value);
```

## Description

This function alternates the value of the game's general variable *name* to *value*. If there is no such variable, it will be created.

*name* – variable's name

*value* – value to be set for the variable

---

## sleep

sleep – suspend the current execution thread temporarily.

## Syntax

**sleep**(number-of-segments);

## Description

This function suspends the work of the current execution thread for the time specified. The time is set in game segments.

The *number-of-segments* must be equal or greater than 1.

The sleep command is necessary to create scripts that are supposed to perform periodical actions within long periods of time (like scripts that realize map scenarios):

```
while 1 do
  sleep(100)
  print("another 100 game segments has passed")
end
```

The important effect of the sleep command is that its execution results in the control's being transferred from the current script thread back to the threads planner of the scripting engine, which, thus, is enabled to let other script threads work and, finally, restore the control back to the game.

For example, if the script cited above is run with the sleep(100) call removed from it, the game will 'freeze': the thread in which the script will be executed, will never restore the control to the scripting engine, which could restore it to the game.

---

## sqrt

sqrt – return the number's square root.

## Syntax

**sqrt**(x);

## Description

This function returns the square root of its only argument, which must be a non-negative number.

---

## startThread

startThread – start a new execution thread.

### Syntax

**startThread**(func);

### Description

This function starts the specified function in a new execution thread.

The *func* parameter must be a script function: *func* cannot be, for example, a number or one of the commands described in this manual (because they are realized in C).

```
local immortal = "immortal-archer";

function eternalLoop()
    while 1 do
        if not exist(%immortal) then
            resurrect(%immortal)
        end
        sleep(1)
    end
end

print("Starting resurrection loop")
startThread(eternalLoop)
```

---

## AddHeroCreatures

AddHeroCreatures – add a stack of creatures to the hero's army.

### Syntax

**AddHeroCreatures**(heroname, creatureID, quantity);

### Description

Adds a *quantity* of creatures of *creatureID* type to the army of the hero *heroname*.

*heroname* is the hero's internal name, set when the map is created.

The command works regardless of where the hero is: he may be absent from the map (being in a town or among the dead (?)).

Only creatures, and not war machines (ballistae, catapultae, etc.) can be added.

---

## AddObjectCreatures

AddObjectCreatures – add a stack of creatures to the object’s army.

### Syntax

**AddObjectCreatures**(objectName, creatureID, quantity);

### Description

This function adds a *quantity* of creatures of *creatureID* type to the army of the object *objectName*. Creatures can be added to any map object that can have an army (including monsters and heroes). Monsters can only have creatures of the same type that the monster already contains added.

*objectName* – the object’s name

*creatureID* – the creatures’ type

*quantity* – the creatures quantity

Only creatures, and not war machines (ballistae, catapultae, etc.) can be added.

---

## BlockGame

BlockGame – block the user interface and the AI work.

### Syntax

**BlockGame**(void);

### Description

The command blocks the user interface and the AI work as well as the camera management. This way, the world comes to a stop, and the script remains as the only source of in-game events. The command also stops the running heroes (while preserving their paths). To unblock the game, the *UnblockGame* command is used. Game blocks are accumulated, which means that the game will be unblocked once the number of calls for *UnblockGame* will be equal to that of *BlockGame*.

---

## CalcHeroMoveCost

CalcHeroMoveCost – calculate the hero's movement cost to the specified point.

### Syntax

```
CalcHeroMoveCost(heroName, x, y, floorID = -1);
```

### Description

This function calculates the hero's movement cost to the specified point, using movement points as the calculating unit. If the floor is not specified, the one in which the hero is currently located is used. This function can only be invoked for heroes controlled by the AI players. Only static objects are taken into account when the path is formed. If the specified point can't be reached, the function returns -1.

*heroName* – the hero's name

*x, y* – the target tile's coordinates

*floorID* – the floor number (-1 by default, which means the floor in which the hero is located)

---

## CanMoveHero

CanMoveHero – determine if the hero can be moved to the specified point

### Syntax

```
CanMoveHero(heroName, x, y, floorID = -1);
```

### Description

This function returns true if the specified point can be reached by the hero. Otherwise it returns false. If the floor is not specified, the one in which the hero is currently located is used. This function can only be invoked for heroes controlled by the AI players.

*heroName* – the hero's name

*x, y* – the target tile's coordinates

*floorID* – the floor number (-1 by default, which means the floor in which the hero is located)

---

## ChangeHeroStat

ChangeHeroStat – change the hero's stats

## Syntax

**ChangeHeroStat**(heroName, statID, delta);

## Description

This function modifies the hero's specified stat to *delta*.

The following stats can be modified: experience, attack, defense, spell power, knowledge, luck, morale, move points, mana points.

*delta* can take on either positive (the stat is increased) or negative (the stat is decreased) values for all stats except experience, since experience can only grow, and for it, *delta* can only be a positive number or zero.

All the stats' values themselves can not be negative. The top values of move points and mana points stats are additionally limited by the move points max and mana points max values. If the modification results in the stat's exceeding the permissible limitations, it is truncated.

*heroName* – the hero's name

*statID* – the stat's ID, which can take on the following values:

STAT\_EXPERIENCE – the hero's experience

STAT\_ATTACK – attack level

STAT\_DEFENCE – defense level

STAT\_SPELL\_POWER – spell power

STAT\_KNOWLEDGE – knowledge

STAT\_LUCK – luck

STAT\_MORALE – morale

STAT\_MOVE\_POINTS – remaining movement points number

STAT\_MANA\_POINTS – current mana points number

*delta* – stat modification value

---

## CreateMonster

CreateMonster– create a monster on the map

## Syntax

**CreateMonster**( *monsterName*,

```
creatureType,  
creaturesCount,  
x,  
y,  
floorID,  
mood= MONSTER_MOOD_AGGRESSIVE,  
courage= MONSTER_COURAGE_CAN_FLEE_JOIN,  
rotation= 0 );
```

## Description

Creates a monster with *creaturesCount* of creatures of the *monsterType* type in the tile ( *x*, *y* ) on the floor *floorID*, setting the name *monsterName* for it. If the specified tile is occupied (there is another object on it, or an object for which it is interactive), the monster is put onto one of the nearby free tiles. The latter two parameters affect the monster's behavior when it encounters a hero (according to the standard logics of this interaction).

*monsterName* – name of the monster, which can then be used in other script commands.

*creatureType* – type of creatures that form the monster.

*creaturesCount* – number of creatures that form the monster.

*x*, *y* – coordinates of the tile onto which the monster is to be placed.

*floorID* – the number of the floor onto which the monster is to be placed.

*mood*, *courage* – these parameters affect the monster's choice of behavior when it encounters a hero, and can take on the following values:

MONSTER\_MOOD\_FRIENDLY,

MONSTER\_MOOD\_AGGRESSIVE,

MONSTER\_MOOD\_HOSTILE,

MONSTER\_MOOD\_WILD and

MONSTER\_COURAGE\_ALWAYS\_JOIN,

MONSTER\_COURAGE\_ALWAYS\_FIGHT,

MONSTER\_COURAGE\_CAN\_FLEE\_JOIN accordingly.

*rotation* – the monster's angle of rotation (0 by default).

---

## DeployReserveHero

DeployReserveHero – puts the hero who has been in the player's reserve onto the map.

## Syntax

**DeployReserveHero**(heroName, x, y, floor);

## Description

When a map is created, some heroes can be reserved after this or that player. Such heroes will not appear in taverns and can't be hired by usual means. To put the hero reserved after a player onto the map, use the *DeployReserveHero* command, setting the hero's name and desired position on the map as its parameters. If the specified tile is inaccessible due to some reason, the hero will be put onto one of the nearby tiles. Reserved heroes who are killed, flee, or disappear from the list of heroes owned by the player for any other reason are put back into the reserve and can be restored to the map by the same command. Their armies are restored to those specified in the editor. To remove a hero from the player's reserve and make him available in other players' taverns use the *UnreserveHero* function.

*heroName* – name of hero reserved after a player.

*x*, *y* – coordinates of the tile onto which the hero is to be put.

*floorID* – the number of the floor onto which the hero is to be put.

---

## EnableAIHeroHiring

EnableAIHeroHiring – turn the AI's hiring heroes in the town's tavern on/off

## Syntax

**EnableAIHeroHiring**(playerID, townName, enable);

## Description

This function lets you allow/prohibit the AI to hire heroes in the tavern of the specified town (by default, this is allowed).

*playerID* – player's ID

*townName* – town name

*enable* – true to turn on, and false to turn off

---

## EnableHeroAI

EnableHeroAI – turn the AI control over the specified hero on/off

## Syntax

**EnableHeroAI**(heroName, enable);

## Description

This function allows turning on/off the AI control over the specified hero. It can only be used in singleplayer mode. When this function is invoked for a human-controlled hero, an error message is generated. If the hero's control status is the same as the function is trying to set it, no action takes place.

*heroName* – the hero's name

*enable* – *true* to turn the AI on, and *false* to turn off

---

## IsObjectExists

IsObjectExists – determine whether the specified object is existent on the map

### Syntax

**IsObjectExists**(objectName);

### Description

This function returns *true* if the object with the specified name is present on the map, and *false* if it isn't. This function can be used for ..., giving OBJECT\_GRAIL as the object's name.

*objectName* – the object's name.

---

## GenerateMonsters

GenerateMonsters – generate monsters on the map

### Syntax

**GenerateMonsters**( *monsterTypeID* ,  
                  *countGroupsMin* ,  
                  *countGroupsMax* ,  
                  *countInGroupMin*,  
                  *countInGroupMax*);

### Description

This function generates monsters at random locations of the adventure map. There are *countGroupsMin* to *countGroupsMax* groups of creatures generated, numbering *countInGroupMin* to *countInGroupMax* each.

*monsterTypeID* – creatures' type ID

*countGroupsMin* – minimal number of groups generated (must be greater or equal to zero)

*countGroupsMax* – maximal number of groups generated (must be greater or equal to *countGroupsMin*)

*countInGroupMin* – minimal number of creatures in the groups (must be greater or equal to zero)

*countInGroupMax* – maximal number of creatures in the groups (must be greater or equal to *countInGroupMin*)

---

---

## GetCurrentPlayer

GetCurrentPlayer – determine the current player

### Syntax

**GetCurrentPlayer**(void);

### Description

This function returns the ID of the player whose turn it is now.

---

---

## GetDate

GetDate – return the current in-game time (day, week, month or day of the week)

### Syntax

**GetDate**(dateTypeID);

### Description

This function returns the current in-game time. The parameter defines what is to be determined: the current day, week, month, or day of the week. By default (when invoked without any parameters), it returns the current day.

*dateTypeID* – type of current time information to be determined

Possible values:

*DAY* – day

*WEEK* – week

*MONTH* – month

*DAY\_OF\_WEEK* – day of week

*ABSOLUTE\_DAY* – same as *DAY*

---

---

## GetHeroCreatures

GetHeroCreatures – determine the number of creatures of the specified type under the hero's command

### Syntax

**GetHeroCreatures**(heroName, creatureID);

### Description

This function returns the number of creatures of the specified type under the specified hero's command.

*heroName* – the hero's name

*creatureID* – creatures' type ID

---

---

## GetHeroLevel

GetHeroLevel – determine the hero's level

### Syntax

**GetHeroLevel**(heroname);

### Description

This function returns the current level of the hero *heroname*.

### Error

- If there is no hero with the name *heroname* ([IsHeroAlive](#)(*heroname*) returns nil/false)
- 
-

# GetHeroStat

GetHeroStat – determine the hero's parameters

## Syntax

**GetHeroStat**(heroName, statID);

## Description

This function returns the value of the specified stat of the hero. The following stats can be determined: experience, attack, defense, spell power, knowledge, luck, morale, move points, mana points. The stats' values are determined, taking into account all the effects (artifacts, etc.)

*heroName* – the hero's name

*statID* – the stat's ID, which can assume the following values:

STAT\_EXPERIENCE – the hero's experience

STAT\_ATTACK – attack level

STAT\_DEFENCE – defense level

STAT\_SPELL\_POWER – spell power

STAT\_KNOWLEDGE – knowledge

STAT\_LUCK – luck

STAT\_MORALE – morale

STAT\_MOVE\_POINTS – remaining movement points number

STAT\_MANA\_POINTS – current mana points number

---

---

# GetObjectCreatures

GetObjectCreatures – determine the number of creatures of the specified type in the object's army

## Syntax

**GetObjectCreature**(objectName, creatureID);

## Description

This function returns the number of creatures of the specified type in the specified object's army. This can be applied to any on-map objects (including monsters and heroes).

*objectName* – the object's name

*creatureID* – creatures' type ID

---

## GetObjectiveProgress

GetObjectiveProgress – determine the progress towards the objective

### Syntax

**GetObjectiveProgress**(objectiveName, playerID = PLAYER\_1);

### Description

This function returns the progress towards the *objectiveName* objective. For objectives that are common for all players, the *playerID* parameter specifies the player for whom the progress is to be determined (if the parameter isn't set, the 1<sup>st</sup> player's progress is determined). For player-specific objectives, the *playerID* parameter is ignored.

The number of steps towards the manually controlled objectives is determined by the number of progress comments as set in the editor. The objectives 'seize the specified on-map objects' and 'destroy the specified neutral armies' have a number of progress steps equal to the number of on-map objects to be seized or the number of armies to be eliminated, accordingly. The other types of objectives have no progress steps and can either be completed or not.

*objectiveName* – objective name

*playerID* – the ID of the player for whom the progress towards objective is to be determined (ignored for objectives that pertain only to this or that player, and equal to *PLAYER\_1* by default)

---

## GetObjectiveState

GetObjectiveState – determine the status of the objective

### Syntax

**GetObjectiveState**(objectiveName, playerID = PLAYER\_1);

### Description

This function returns the status of the objective *objectiveName* for the specified player. For player-specific objectives, the *playerID* parameter is ignored. For common objectives, if the *playerID* parameter is specified, it indicates the player for whom the objective's status is to be determined; otherwise the status for the 1<sup>st</sup> player is returned.

*objectiveName* – objective name

*playerID* – the ID of the player for whom the status of the objective is to be determined (ignored for objectives that pertain only to this or that player, and equal to *PLAYER\_1* by default)

List of possible statuses of the objectives:

OBJECTIVE\_SCENARIO\_INFO – a special status which means that this objective is in fact the current map's scenario description.

OBJECTIVE\_UNKNOWN – the objective is unknown to the player. If the visibility flag is specified for the objective (see the functions *GetObjectiveState* / *SetObjectiveState*), the objectives interface only contains a vague description of the task.

OBJECTIVE\_ACTIVE – the player is now performing the objective.

OBJECTIVE\_COMPLETED – the objective is completed.

OBJECTIVE\_FAILED – the objective is failed.

---

---

## GetObjectOwner

GetObjectOwner – determine the belonging of the specified on-map objects

### Syntax

**GetObjectOwner**(*objectName*);

### Description

This function returns the ID of the player who owns the specified object. If the object belongs to no one, it returns *PLAYER\_NONE*.

*objectName* – the object's name

---

---

## GetObjectsInRegion

GetObjectsInRegion – determine what objects are present in the specified region

## Syntax

**GetObjectsInRegion**(regionName, objectType);

## Description

This function returns the names of the on-map objects of the specified type that are within the region. Objects with no names are ignored.

*regionName* – name of the region

*objectType* – type of the objects to be checked for presence in the region

To date, the function only works with one type of objects:

OBJECT\_HERO – heroes

---

---

## GetObjectPosition

GetObjectPosition – determine the object's position on the map

## Syntax

**GetObjectPosition**(objectName);

## Description

This function returns three values: the x and y coordinates of the tile in which the object is positioned (or, if it occupies more than one tile, the function returns the position of its center) and the number of the floor in which it stands. This function can be used for ..., giving OBJECT\_GRAIL as the object's name.

*objectName* – the object's name

---

---

## GetPlayerHeroes

GetPlayerHeroes – return the names of the heroes belonging to the player

## Syntax

**GetPlayerHeroes**(playerID);

## Description

This function returns an array that contains the names of the heroes belonging to the specified player.

*playerID* – the player's number

---

## GetPlayerResource

GetPlayerResource – get the amount of the specified player's resources.

### Syntax

**GetPlayerResource**(player, resourceKind);

### Description

This function returns the amount of resources of the type *resourceKind* possessed by the player *player*.

*player* — the player's number from 1 to 8. Global constants PLAYER\_1 to PLAYER\_8 can be used instead of the numbers.

*resourceKind* – a number from 0 to 6, or one of the constants: WOOD, ORE, MERCURY, CRYSTAL, SULFUR, GEM, or GOLD.

The command works for both active and failed players. Requesting the amount of resources owned by a player who did not take part in the current game is an error.

### Error

- If the arguments don't meet the values permissibility criteria;
  - if the specified player does not take part in the current game
- 

## GetTownBuildingLevel

GetTownBuildingLevel – determine the level of the building in the town

### Syntax

**GetTownBuildingLevel**(townName, buildingID);

### Description

This function returns the level of the specified building in the town. Level 0 means that the building has not been erected.

*townName* – town name

*buildingID* – the ID of the building type, which can take on the following values:

TOWN\_BUILDING\_TOWN\_HALL,

TOWN\_BUILDING\_FORT,

TOWN\_BUILDING\_MARKETPLACE,

TOWN\_BUILDING\_SHIPYARD,

TOWN\_BUILDING\_TAVERN,

TOWN\_BUILDING\_BLACKSMITH,

TOWN\_BUILDING\_MAGIC\_GUILD,

TOWN\_BUILDING\_DWELLING\_1,

TOWN\_BUILDING\_DWELLING\_2,

TOWN\_BUILDING\_DWELLING\_3,

TOWN\_BUILDING\_DWELLING\_4,

TOWN\_BUILDING\_DWELLING\_5,

TOWN\_BUILDING\_DWELLING\_6,

TOWN\_BUILDING\_DWELLING\_7,

TOWN\_BUILDING\_GRAIL,

TOWN\_BUILDING\_WONDER,

TOWN\_BUILDING\_HAVEN\_TRAINING\_GROUNDS,

TOWN\_BUILDING\_HAVEN\_MONUMENT\_TO\_FALLEN\_HEROES,

TOWN\_BUILDING\_HAVEN\_HOSPITAL,

TOWN\_BUILDING\_HAVEN\_STABLE,

TOWN\_BUILDING\_HAVEN\_FARMS,

TOWN\_BUILDING\_INFERNO\_INFERNAL\_LOOM,

TOWN\_BUILDING\_INFERNO\_ORDER\_OF\_FIRE,

TOWN\_BUILDING\_INFERNO\_HALLS\_OF\_HORROR,

TOWN\_BUILDING\_INFERNO\_SACRIFICIAL\_PIT,  
TOWN\_BUILDING\_DUNGEON\_ALTAR\_OF\_ELEMENTS,  
TOWN\_BUILDING\_DUNGEON\_RITUAL\_PIT,  
TOWN\_BUILDING\_DUNGEON\_TRADE\_GUILD,  
TOWN\_BUILDING\_DUNGEON\_TREASURE\_DIG\_SITE,  
TOWN\_BUILDING\_DUNGEON\_HALL\_OF\_INTRIGUE,  
TOWN\_BUILDING\_ACADEMY\_LIBRARY,  
TOWN\_BUILDING\_ACADEMY\_ARCANE\_FORGE,  
TOWN\_BUILDING\_ACADEMY\_ARTIFACT\_MERCHANT,  
TOWN\_BUILDING\_ACADEMY\_TREASURE\_CAVE,  
TOWN\_BUILDING\_ACADEMY\_ELEMENTAL\_ENCLAVE,  
TOWN\_BUILDING\_PRESERVE\_AVENGERS\_BROTHERHOOD,  
TOWN\_BUILDING\_PRESERVE\_MYSTIC\_POND,  
TOWN\_BUILDING\_PRESERVE\_SPARKLING\_FOUNTAINS,  
TOWN\_BUILDING\_PRESERVE\_BLOOMING\_GROVE,  
TOWN\_BUILDING\_PRESERVE\_TREANT\_SAMPLING,  
TOWN\_BUILDING\_NECROMANCY\_AMPLIFIER,  
TOWN\_BUILDING\_NECROMANCY\_UNHOLY\_TEMPLE,  
TOWN\_BUILDING\_NECROMANCY\_UNEARTHED\_GRAVES,  
TOWN\_BUILDING\_NECROMANCY\_DRAGON\_TOMBSTONE,  
TOWN\_BUILDING\_NECROMANCY\_SHROUD\_OF\_DARKNESS.

NB: Giving a building that is specific for another race's towns can cause erroneous results.

---

---

## GetTownBuildingLimitLevel

GetTownBuildingLimitLevel – determine the level limitations for a building in the town.

## Syntax

**GetTownBuildingLimitLevel**(townName, buildingID);

## Description

This function returns the possible level limitation of the specified building in the town. Level 0 means that the building can not been erected.

*townName* – town name

*buildingID* – the building type ID (see the description of *GetTownBuildingLevel* for the list of the possible values).

---

---

## GetTownBuildingMaxLevel

GetTownBuildingMaxLevel – determine the maximal level of the building in the town

## Syntax

**GetTownBuildingMaxLevel**(townName, buildingID);

## Description

This function returns the maximal possible level of the specified building in the town. Level 0 means that the building can not been erected.

*townName* – town name

*buildingID* – the building type ID (see the description of *GetTownBuildingLevel* for the list of the possible values).

---

---

## GetTownHero

GetTownHero – return the hero garrisoned in the town

## Syntax

**GetTownHero**(townName);

## Description

This function returns the name of the hero garrisoned in the specified town, or nil if there is no hero in the garrison.

*townName* – town name

---

---

## GiveArtefact

GiveArtefact – give an artifact to the hero

### Syntax

**GiveArtefact**(*heroname*, *artefactID*, [*bindToHero* = 0]);

### Description

Grants an artifact *artefactID* to the hero *heroname*.

*artefactID* – the number of the artifact from 0 to 55, or a character constant from the list

*bindToHero* – bind the artifact to the hero (making it impossible to hand over the artifact to another hero)

### Error

- if there is no hero with the name *heroname* ([IsHeroAlive](#)(*heroname*) returns nil/false)
- 
- 

## LevelUpHero

LevelUpHero – gives the hero as many experience points as needed to gain the next level

### Syntax

**LevelUpHero**(*heroName*);

### Description

This function gives the hero *heroName* as many experience points as he or she needs to gain the next level. This function returns `true` if it has been possible to grant a level to the hero, and `nil` if it's impossible (the hero having the maximal possible level already).

*heroName* – the hero's name

---

---

## GiveHeroSkill

GiveHeroSkill – give the specified basic skill to the hero

## Syntax

**GiveHeroSkill**(heroName, skillID);

## Description

This function tries to give a specified basic skill to the hero. It returns `false` if this is impossible (for ex., the hero already has ‘EXPERT’ mastery of this skill, or there is no space on the bar for it), and otherwise it returns `true`.

*heroName* – the hero’s name

*skillID* – the skill ID

---

---

# GiveHeroWarMachine

GiveHeroWarMachine – give the hero a war machine of the specified type

## Syntax

**GiveHeroWarMachine**(heroName, warMachineType);

## Description

This function tries to give the specified war machine to the hero. It returns `nil` if the hero already has such a war machine, and otherwise it returns `non nil`.

*heroName* – the hero’s name

*warMachineID* – war machine type ID

---

---

# HasArtefact

HasArtefact – determine whether the hero has the artifact

## Syntax

**HasArtefact**(heroname, artefactID);

## Description

This function returns whether the hero *heroname* has the artifact *artefactID*.

*artefactID* – the number of the artifact from 0 to 54 (the artifacts [description](#) at the H5 web site says, ‘there are 54 artifacts, and one of them can become double, so there will be 55’, but in fact there are only 53 of them in the game).

We should also (maybe) fix character constants for the artifacts, to use `HasArtefact("Agrael", SWORD_OF_RUINS)` instead of `HasArtefact("Agrael", 0)`.

## Error

- if there is no hero with the name *heroname* ([IsHeroAlive](#)(*heroname*) returns nil/false)
- 
- 

## HasBorderguardKey

`HasBorderguardKey` – determine whether the player has the key of the specified color

### Syntax

`HasBorderguardKey(player, color);`

### Description

This function returns not `nil` if the player has the key of the specified color, and `nil` if not.

*player* – the player’s number

*color* – the color of the key; can take on the following values:

RED\_KEY, BLUE\_KEY, GREEN\_KEY, YELLOW\_KEY, ORANGE\_KEY, TEAL\_KEY, PURPLE\_KEY, TAN\_KEY.

---

---

## HasHeroSkill

`HasHeroSkill` – checks if the hero has the specified skill

### Syntax

`HasHeroSkill(heroName, skillID);`

### Description

This function returns `true` if the hero has the specified skill (by his or her own, or granted by an artifact), and otherwise it returns `false`.

*heroName* – the hero’s name

*skillID* – the skill ID

---

## HasHeroWarMachine

HasHeroWarMachine - determines whether the hero has a war machine of the specified type

### Syntax

**HasHeroWarMachine**(heroName, warMachineType);

### Description

It returns not `nil` if the hero has a war machine of the specified type, and `nil` otherwise.

*heroName* – the hero's name

*warMachineID* – war machine type ID

---

## IsHeroAlive

IsHeroAlive – is the hero alive?

### Syntax

**IsHeroAlive**(heroname);

### Description

This function returns if there is a hero *heroname*, and if yes, if this hero is alive.

A hero is considered as alive if belongs to any of the active players (for whom [GetPlayerState\(\)](#) returns true). It does not matter where the hero is during the enquiry: in the surface, in the Underworld, in the town, or in a boat.

*heroname* is an internal scripting name of the hero (not only the name that is specified when the map is created).

---

## IsHeroLootable

IsHeroLootable – determine whether it is possible to loot artifacts from the hero by defeating him or her.

## Syntax

**IsHeroLootable**(heroName);

## Description

It returns not `nil` if the hero's artifacts are delivered to the winner after the hero is defeated, or `nil` if they remain with the defeated hero.

*heroName* – the hero's name

---

## IsObjectEnabled

**IsObjectEnabled** – determine whether the interactive object interacts with the hero standardly

### Syntax

**IsObjectEnabled**(objectName);

### Description

If the function returns `true`, the interactive object behaves standardly when a hero comes to it. If the function returns `false`, when the hero comes to this object, nothing happens but the invocation of the trigger handler function, if such function has been set. By default, an interactive object behaves standardly.

*objectName* – the interactive object's name.

---

## IsObjectInRegion

**IsObjectInRegion** – determine whether the object is within the specified region

### Syntax

**IsObjectInRegion**(objectName, regionName);

### Description

This function returns `true`, if the object is within the specified region, and `false` if not.

NB: the function works correctly only for one-tile objects.

*objectName* – the object's name

*regionName* – the region's name

---

---

## IsObjectiveVisible

IsObjectiveVisible – determine whether the objective is shown in the specified player's interface

### Syntax

**IsObjectiveVisible**(objectiveName, playerID = PLAYER\_1);

### Description

This function returns `true`, if the objective is shown in the specified player's interface, and `false` if not. For player-specific objectives, the *playerID* parameter is ignored. For common objectives, if the *playerID* parameter is given, it sets the player for whom the objective's visibility is to be determined; if not, the function determines whether the 1<sup>st</sup> player sees the objective.

*objectiveName* – objective name

*playerID* – the player's ID (for player-specific objectives, the parameter is ignored, and it is equal to `PLAYER_1` by default).

---

---

## IsObjectVisible

IsObjectVisible – determine whether the object is visible to the player

### Syntax

**IsObjectVisible**(playerID, objectName);

### Description

This function returns `true`, if the object is visible to the player, and `false` if not.

NB: the function works correctly only for one-tile objects.

*playerID* – the player's ID

*objectName* – the object's name

---

---

## IsRegionBlocked

IsRegionBlocked – determine whether the region is blocked for the maneuvers of the specified player's heroes

### Syntax

**IsRegionBlocked**(regionName, playerId);

### Description

This function returns not `nil` if the region is blocked for the maneuvers of the specified player's heroes, and `nil` if not.

*regionName* – the region's name

*playerID* – the player's ID

---

---

## KnowHeroSpell

KnowHeroSpell – checks whether the hero knows the spell

### Syntax

**KnowHeroSpell**(heroName, spell);

### Description

This function returns `true`, if the hero knows the specified spell (regardless of whether this knowledge was learned or granted by an artifact), otherwise it returns `false`.

*heroName* – the hero's name

*spell* – the spell ID

See the appendix for the list of spell IDs.

---

---

## Loose

Loose – the human player loses the game when the function is invoked.

### Syntax

**Loose**(void);

## Description

This the function should only be used in the single-player mode. It generates an error if invoked in the multiplayer mode. When the function is invoked, the human player loses immediately.

---

---

## MarkObjectAsVisited

MarkObjectAsVisited – mark the inoperative interactive object as ‘visited’ by the specified hero

### Syntax

**MarkObjectAsVisited**(objectName, heroName);

### Description

The command should be invoked when the hero interacts with an interactive object that has been made inoperative by the *SetObjectEnabled* command.

*objectName* – the interactive object’s name

*heroName* – the name of the hero who interacts with the object

---

---

## MessageBox

MessageBox – send a message to the screen

### Syntax

**MessageBox**(messageName, callback = "");

### Description

This function sends a message to the screen, which is a dialogue box with the only button ‘OK’. The *callback* parameter (if it is given) sets the name of the function to be invoked when the user clicks on the ‘OK’ button.

*messageName* – name of the message in the resources base

*callback* – name of the function to be invoked when the user clicks on the ‘OK’ button

---

---

# MoveCamera

MoveCamera – move the camera to the specified point of the map

## Syntax

**MoveCamera**(*x*, *y*, *floorID*, *zoom* = 50, *pitch* =  $\pi/2$ , *yaw* = 0, *noZoom* = 0, *noRotate* = 0);

## Description

This function moves the camera to the specified position and sets the specified values of zoom and angle.

*x*, *y*, *floorID* – these parameters specify the tile of the map on which the camera is to be focused

*zoom* – this parameter determines how far will the camera be set from the specified tile

*pitch* – the camera's angle (0 – the camera watches horizontally,  $\pi/2$  – the camera watches vertically down)

*yaw* – the camera's rotation angle (0 – north)

*noZoom* – set to 1 for the camera not to change its zoom when it moves

*noRotate* – set to 1 for the camera not to rotate when it moves

---

# MoveHero

MoveHero – orders the hero to move to a specified point

## Syntax

**MoveHero**(*heroName*, *x*, *y*, *floorID* = -1);

## Description

This function orders the hero to move to the specified location. If the floor number is not specified, the hero is supposed to go to the point with the specified coordinates on the floor he or she already stands on. This function can only be invoked for heroes controlled by the AI players, with their AI turned off. If the specified location is inaccessible, the function generates an error. If the hero's movement points are depleted en route, the movement will be continued in the next turn.

*heroName* – the hero's name

*x*, *y* – the target tile's coordinates

*floorID* – the floor number (-1 by default, which means the floor in which the hero is located)

---

## MoveHeroRealTime

MoveHeroRealTime – make the hero move to the specified point

### Syntax

**MoveHeroRealTime**(heroName, x, y, floorID = -1);

### Description

This function orders the hero to move to the specified location. If the floor number is not specified, the hero is supposed to go to the point with the specified coordinates on the floor he or she already stands on. If the specified location is inaccessible, the function generates an error. The movement begins immediately after the command has been invoked, regardless of whose turn it is now. The script continues its work immediately after the command is executed, without waiting for the hero to reach the destination. The user interface and the AI's actions are blocked while the hero moves. If the hero's movement points are depleted en route, or the hero encounters an obstacle, he or she stops without interacting with the object (which, in particular, means that the hero is unable to use teleports, boats, etc.).

*heroName* – the hero's name

*x*, *y* – the target tile's coordinates

*floorID* – the floor number (-1 by default, which means the floor in which the hero is located)

---

## GetAllNames

GetAllNames – give the list of on-map objects' names

### Syntax

**GetAllNames**(filter = 0);

### Description

This function returns a string with the list of on-map objects. The unit names are separated by the space.

The *filter* parameter sets the type of objects whose names are to be included in the result. The possible values of *filter*:

- 0 – names of the active players' heroes

---

---

## OpenCircleFog

OpenCircleFog – remove the warfog within the specified cricle

### Syntax

**openCircleFog**(*x*, *y*, *floorID*, *range*, *playerID*);

### Description

This function opens the player's fog of war at the floor *floorID*, within a circle that has its center at(*x*, *y*) and radius *range*.

*player* – the player's ID

*x*, *y* – the circle's center coordinates

*floorID* – floor number

*range* – circle radius

---

---

## OpenRegionFog

OpenRegionFog – remove the warfog within the specified region

### Syntax

**openRegionFog**(*player*, *regionName*);

### Description

This function opens the player's fog of war within the specified region.

*player* – the player's ID

*regionName* – the region's name

---

---

## Play2DSound

Play2DSound – play 2D sound

## Syntax

**Play2DSound**(soundName);

## Description

Play the 2D sound with the specified name.

If the base determines the sound as being cyclic, the function returns the number which is an ID of the sound that can then be sent to the function *StopPlaySound* to stop the sound.

If the sound is determined as non-cyclic, the function returns -1.

*soundName* – the sound's name

---

---

## Play3DSound

Play3DSound – play 3D sound

## Syntax

**Play3DSound**(soundName, x, y, floor);

## Description

Play the 3D sound with the specified name. The sound's source is in the tile with coordinates  $x$  и  $y$ , at the *floor*.

If the base determines the sound as being cyclic, the function returns the number which is an ID of the sound that can then be sent to the function *StopPlaySound* to stop the sound.

If the sound is determined as non-cyclic, the function returns -1.

*soundName* – the sound's name

---

---

## PlayObjectAnimation

PlayObjectAnimation – play an animation at the on-map object

## Syntax

**PlayObjectAnimation**(objectName, animName, action);

## Description

Play *animName* animation at the specified on-map object.

The *action* determines how the animation is played. There are the following action types in the game:

INVISIBLE – (special) the object becomes invisible

IDLE – the animation is played cyclically, until it is replaced by another animation

ONESHOT\_STILL – the animation is played once, and after it is finished, the object remains in the posture it took in the last frame of the animation

ONESHOT – the animation is played once, and after it the object's standard idle animation is started

NON\_ESSENTIAL – same as ONESHOT, but the animation is only played in case if the object is not 'busy', which means that nothing plays on it except the IDLE animation

animName – the animation's character name ("idle00", "attack00", "hit", etc.), which should be known in advance. If the object has no animation with the name animName, the function does nothing.

---

---

## random

random – return a random number

### Syntax

**random**(top);

### Description

This function returns a random whole number from within the range from zero to `top - 1`. The function's argument must be a positive integer.

---

---

## RazeTown

RazeTown – raze a town on the map

### Syntax

**RazeTown**(townName);

### Description

Removes the specified town from the map and replaces it with a static object – razed town. The 'razed town' object is determined by the *razed* field in *AdvMapTownShared*. If this field is empty, the town can't be razed. The set of blocking tiles of the razed town must be the same as

those of the original one's. The razed town, as any other static object, must have no interactive tiles.

*townName* – town name

Warning!

The modifications made in the "Shared" properties of an object will affect all the objects of that kind in the game. If you want your modification to affect to be effective only on a specific map follow these steps:

- 1) Place on the map the desired object.
- 2) Go to the Shared field of that object.
- 3) In the Objects list create a new folder (Right click-New Folder) with the name of your map.
- 4) Place in this folder a copy of the object you want to modify.
- 5) Make the desired modifications to the copy.

For the modifications to be in effect make sure that the that path of the shared property leads to the copy.

---

---

## RegionToPoint

RegionToPoint – determine the coordinates and number of floor of the punctual region

### Syntax

**RegionToPoint**(regionName);

### Description

This function returns the *x* and *y* coordinates, and the floor number, in which the singular, one-tile region is located. If a non-singular region is passed to the function, an error is generated.

*regionName* – the region's name

---

---

## RemoveArtefact

RemoveArtefact – remove the artifact from the hero

### Syntax

**RemoveArtefact**(*heroname*, *artefactID*);

## Description

Remove the artifact *artefactID* from the hero *heroname*.

*artefactID* – the number of the artifact from 0 to 55, or a character constant from the list

## Error

- if there is no hero with the name *heroname* ([IsHeroAlive](#)(*heroname*) returns nil/false)
  - if the hero has no such artifacts (*advmap.HasArtefact*(*heroName*, *artefactID*) returns nil/false)
- 
- 

## RemoveHeroCreatures

RemoveHeroCreatures – remove creatures from the hero's army

### Syntax

**RemoveHeroCreatures**(*heroname*, *creatureID*, *quantity*);

### Description

Remove a *quantity* of creatures of the *creatureID* type from the army of the hero *heroname*. If the number of creatures of the specified type in the hero's army is lower than that set in the *quantity* parameter, all such creatures are removed from his or her army. If there are no other creatures except those to be removed in the hero's army, the command removes them all except one.

*heroname* – the hero's name.

The command also works for inactive heroes (who are in the taverns, prisons, etc.)

Only creatures, and not war machines (ballistae, catapultae, etc.) can be removed.

*creatureID* – the creatures' type

*quantity* – the creatures quantity

---

---

## RemoveHeroWarMachine

RemoveHeroWarMachine – remove the war machine of the specified type from the hero

## Syntax

**RemoveHeroWarMachine**(heroName, warMachineType);

## Description

This function tries to take the specified war machine from the hero. If the hero does not have it, or it's impossible to remove it, the function returns `nil`, otherwise it returns `not nil`.

*heroName* – the hero's name

*warMachineID* – war machine type ID

NB: A hero always has a catapult, and it can't be removed.

---

---

## RemoveObject

RemoveObject – remove an object from the adventure map.

### Syntax

**RemoveObject**(objectName);

### Description

This function removes the object with the specified name from the adventure map.

*objectName* – the object's name

---

---

## RemoveObjectCreatures

RemoveObjectCreatures – remove creatures from the object's army

### Syntax

**RemoveObjectCreatures**(objectName, creatureID, quantity);

### Description

Remove a *quantity* of creatures of the *creatureID* type from the army of the object *objectName*. If the number of creatures of the specified type in the object's army is lower than that set in the *quantity* parameter, all such creatures are removed from its army. The command can be applied to all on-map objects that have armies (including monsters and heroes). If the object is a hero and there are no other creatures except those to be removed in the hero's army,

the command removes them all except one. If the object is a monster and all creatures that form it are removed, the monster is removed from the map.

*objectName* – the object's name

*creatureID* – the creatures' type

*quantity* – the creatures quantity

Only creatures, and not war machines (ballistae, catapultae, etc.) can be removed.

---

## ResetHeroCombatScript

ResetHeroCombatScript – resets the script that would have been started when combating this hero

### Syntax

**ResetHeroCombatScript**(heroName);

### Description

This function tells that no script is to be run when combating this hero. If the combat takes place in a location for which another combat script is specified (town, garrison, etc.), that script will be run.

*heroName* – the hero's name

---

## ResetObjectFlashlight

ResetObjectFlashlight – remove the point-light source from the on-map object

### Syntax

**ResetObjectFlashlight**(objectName);

### Description

The command removes the additional light source, added by the *SetObjectFlashlight* command, from the selected object.

*objectName* – the object's name.

---

---

## SetAIHeroAttractor

SetAIHeroAttractor – set the goal for the specified AI hero

### Syntax

**SetAIHeroAttractor**(objectName, heroName, priority);

### Description

This function modifies the house evaluation for the specified AI hero.

NB: Evaluation modification can be removed if 0 priority is set.

NB: the function works correctly only for immobile objects.

*objectName* – the object's name

*heroName* – the hero's name

*priority* – goal priority (withn the range of -1 to 2).

-1 – house evaluation decreased greatly.

0 – house evaluation remains unchanged

1 – house evaluation increases greatly. But the AI will not risk its heroes for claiming this house.

2 – house is evaluated equally to winning the game. The AI will ignore all dangers for seizing this house.

---

---

## SetAIPlayerAttractor

SetAIPlayerAttractor – set the goal for all the AI player's heroes

### Syntax

**SetAIPlayerAttractor**(objectName, playerID, priority);

### Description

This function modifies the house rating for the all the heroes of the specified AI player.

NB: Evaluation modification can be removed if 0 priority is set.

NB: the function works correctly only for immobile objects.

*objectName* – the object's name

*playerID* – the player's ID

*priority* – goal priority (withn the range of -1 to 2).

-1 – house evaluation decreased greatly.

0 – house evaluation remains unchanged

1 – house evaluation increases greatly. But the AI will not risk its heroes for claiming this house.

2 – house is evaluated equally to winning the game. The AI will ignore all dangers for seizing this house.

---

---

## SetCombatLight

SetCombatLight – modify the illumination in all the map's combat arenas

### Syntax

**SetCombatLight**(lightName);

### Description

This function changes the ambient light source for all the combat arenas of the map for that specified in the command's parameters.

*lightName* – reference to resource of the the ambient light source from the resources base

---

---

## SetHeroCombatScript

SetHeroCombatScript – tell which script is to be run when combating this hero

### Syntax

**SetHeroCombatScript**(heroName, scriptName);

### Description

This function lets you tell which script is to be run when combating this hero The script is only run if the hero is under the AI player's control, and no other script is set for the location where

the combat is taking place (town, garrison, etc.). The script is reset automatically when the player somehow loses this hero.

*heroName* – the hero's name

*scriptName* – the reference to the script

---

---

## SetHeroLootable

SetHeroLootable – set whether it is possible to take the hero's artifacts by defeating him or her

### Syntax

```
SetHeroLootable(heroName, enable);
```

### Description

The *enable* parameter determines whether the heroes' artifacts will be given to his or her winner (not `nil`) or remain with the defeated hero (`nil`).

*heroName* – the hero's name

*enable* – the parameter determines whether the heroes' artifacts are taken from him or her after defeat

---

---

## SetAmbientLight

SetAmbientLight – change the ambient light at the specified floor of the map

### Syntax

```
SetAmbientLight(floorID, lightName, fade = false, time = 1);
```

### Description

This function sets a new source of ambient light at the specified floor of the map. The list of available light sources is specified in the map description.

*floorID* – the number of the floor in which the light is changed

*lightName* – name of the ambient light source

*fade* – the parameter determines whether the lighting must change gradually (*fade* = true) or immediately (*fade* = false) (the old and new light sources must only differ in color components)

*time* – how much time the change will take

---

---

## SetObjectEnabled

SetObjectEnabled – turn on/off the interactive object's standard interaction with heroes

### Syntax

**SetObjectEnabled**(objectName, enable);

### Description

The command allows turning on/off the interactive object's standard behavior when a hero comes to it. By default, an interactive object behaves standardly. If the function is invoked with *enable* parameter equal to false, when the hero comes to this object, nothing happens but the invocation of the trigger handler function, if such function has been set.

*objectName* – the interactive object's name

*enable* – the flag that sets whether the object will behave standardly when a hero comes to it

---

---

## SetObjectiveProgress

SetObjectiveProgress – set the progress towards objective

### Syntax

**SetObjectiveProgress**(objectiveName, step, playerId = PLAYER\_1);

### Description

This function sets the progress towards the *objectiveName* objective. For objectives that are common for all players, the *playerID* parameter specifies the player for whom the progress is to be set (if the parameter isn't set, the 1<sup>st</sup> player's progress is set). For player-specific objectives, the *playerID* parameter is ignored.

The progress can only be managed for the manually controlled objectives. The number of steps towards the manually controlled objectives is determined by the number of progress comments as set in the editor.

*objectiveName* – objective name

*step* – step of the progress

*playerID* – the ID of the player for whom the progress is to be set (for player-specific objectives, the parameter is ignored, and it is equal to PLAYER\_1 by default).

---

## SetObjectiveState

SetObjectiveState – change the objective's status

### Syntax

**SetObjectiveState**(*objectiveName*, *state*, *playerID* = PLAYER\_1);

### Description

This function changes the status of the objective *objectiveName* for the specified player. For player-specific objectives, the *playerID* parameter is ignored. For common objectives, if the *playerID* parameter is specified, it indicates the player for whom the objective's status is to be changed; otherwise the status for the 1<sup>st</sup> player is changed.

*objectiveName* – objective name

*state* – status in which the objective is to be set

*playerID* – the ID of the player for whom the objective's status is to be set (for player-specific objectives, the parameter is ignored, and it is equal to PLAYER\_1 by default).

The table of permissible changeovers:

OBJECTIVE\_SCENARIO\_INFO -> none (the task is in fact a scenario description)  
OBJECTIVE\_UNKNOWN -> OBJECTIVE\_ACTIVE (if the objective's activation does not depend on other objectives' completion) and OBJECTIVE\_FAILED (if the objective is determined as manually controlled).

OBJECTIVE\_ACTIVE -> OBJECTIVE\_COMPLETED, OBJECTIVE\_FAILED (if the objective is determined as manually controlled).

OBJECTIVE\_COMPLETED -> OBJECTIVE\_ACTIVE and OBJECTIVE\_FAILED (if the objective is specified as one that can lose the 'completed' status, and it is determined as manually controlled).

OBJECTIVE\_FAILED -> none

NB: When an objective is set to OBJECTIVE\_ACTIVE status, it becomes visible to the player automatically (see the *IsObjectiveVisible* / *SetObjectiveVisible* functions).

---

---

## SetObjectiveVisible

SetObjectiveVisible – show/hide the objective in the specified player's interface

### Syntax

```
SetObjectiveVisible(objectiveName, enable, playerId = PLAYER_1);
```

### Description

The objective *objectiveName* will be show in the player's interface if the *enable* parameter is equal to `true`, or it will be hidden if the parameter is equal to `false`. For player-specific objectives, the *playerID* parameter is ignored. For common objectives, if the *playerID* parameter is given, it sets the player for whom the objective's visibility is to be determined; if not, the function determines whether the 1st player sees the objective.

*objectiveName* – objective name

*playerID* – the player's ID (for player-specific objectives, the parameter is ignored, and it is equal to `PLAYER_1` by default).

---

---

## SetObjectFlashlight

SetObjectFlashlight – attach a point-light source to the on-map object

### Syntax

```
SetObjectFlashlight(objectName, lightName);
```

### Description

The command attaches an additional point-light source to the selected on-map object. The list of light sources that can be used is specified in the map description (the *resources->pointLights* field). Their names are specified there as well.

*objectName* – the object's name.

*lightName* – the light source name (specified in the map properties)

---

---

## SetObjectOwner

SetObjectOwner – change the belonging of the specified on-map objects

## Syntax

**SetObjectOwner**(objectName, playerId);

## Description

This command changes the belonging of an on-map object. It can only be used for static objects that can have owners, and heroes. If the hero was in the reserve, he or she will be removed from it. Heroes can not become neutral.

*objectName* – the object's name

*playerID* – the ID of the player who is the object's new owner

---

---

## SetObjectPosition

SetObjectPosition – move the object

## Syntax

**SetObjectPosition**(objectName, x, y, floor = -1);

## Description

This function instantly moves the object into the position specified by the tile coordinates and floor number. If the *floor* parameter is not specified in the invocation of the function, the object's current floor will be used by default. Only mobile objects can be moved. If the target position is inaccessible, occupied with another object, or is an interactive tile of another object, the function generates an error.

*objectName* – the object's name

*x*, *y* – the target tile's coordinates

*floor* – floor number (equal to -1 by default, which means 'same')

---

---

## SetPlayerResource

GetPlayerResource – set the amount of the specified player's resources.

## Syntax

**SetPlayerResource**(player, resourceKind, quantity);

## Description

Sets the new amount of resource type *resourceKind* for the *player*.

*player* – is the player's number from 1 to 8. Global constants PLAYER\_1 to PLAYER\_8 can be used instead of the numbers.

*resourceKind* – a number from 0 to 6, or one of the constants: WOOD, ORE, MERCURY, CRYSTAL, SULFUR, GEM, or GOLD.

*quantity* can't be negative.

The command works for both active and failed players. Changing the amount of resources owned by a player who did not take part in the current game is an error.

## Error

- If the arguments don't meet the values permissibility criteria;
- if the specified player does not take part in the current game

---

# SetPlayerStartResources

GetPlayerResource – set the initial amount of the specified player's resources

## Syntax

**SetPlayerStartResources**(player, wood, ore, mercury, crystal, sulfur, gem, gold);

## Description

Sets the new initial amount of resource type *resourceKind* for the *player*.

*player* is the player's number from 1 to 8. Global constants PLAYER\_1 to PLAYER\_8 can be used instead of the numbers.

*wood, ore, mercury, crystal, sulfur, gem, gold* – initial resource values, which can't be below zero.

The command is equivalent to invoking SetPlayerResource for each type of resources, except for its effect on the mission results interface.

The command works for both active and failed players. Changing the amount of resources owned by a player who did not take part in the current game is an error.

## Error

- If the arguments don't meet the values permissibility criteria;
  - if the specified player does not take part in the current game
- 
- 

## SetRegionBlocked

SetRegionBlocked – set on/off the blocking of the region for the maneuvers of the players' heroes

### Syntax

**SetRegionBlocked**(regionName, status, playerId = -1);

### Description

This function blocks or unblocks the region for the specified players' maneuvers, depending on the *status* parameter. If the *playerID* parameter is set, the region is only blocked for this player's heroes; if not, it will be blocked for all heroes.

*regionName* – the region's name

*status* – the flag to determine whether the blocking is set on or off

*playerID* – the player's ID (equal to -1, which means all players, by default)

---

---

## SetTownBuildingLimitLevel

SetTownBuildingLimitLevel – set the level limitations for a building in the town

### Syntax

**SetTownBuildingLimitLevel**(townName, buildingID, limit);

### Description

This function sets the possible level limitation of the specified building in the town. Level 0 means that the building can not be erected. The set limitation must not be lower than the specified building's current level.

*townName* – town name

*buildingID* – the building type ID (see the description of *GetTownBuildingLevel* for the list of the possible values)

*limit* – level limitation for the building in the town

---

---

## SetWarfogBehaviour

SetWarfogBehaviour – turn on/off the player's heroes' removing the fog of war

### Syntax

```
SetWarfogBehaviour(onLand, onSea);
```

### Description

*onLand*(1/0) – whether the player's hero will remove the fog of war when moving *on land*.

*onSea*(1/0) – whether the player's hero will remove the fog of war when moving *on sea*.

---

---

## ShowFlyingSign

ShowFlyingSign – show a message flying off the object

### Syntax

```
ShowFlyingSign(messageName, objectName, targetPlayerID = -1, time = 1.0);
```

### Description

This function shows a message that flies off the object with name *objectName*. The message is specified according to the resource name in the base (*messageName*). Messages can be shown for one player only, or for all of them. To show a message to one player only, the *targetPlayerID* parameter must contain this player's ID, and to show it to all players it must be -1. The message's flying time is *time* (one second by default).

*messageName* – name of the message in the resources base

*objectName* – the name of the object over which the message will appear

*targetPlayerID* – the ID of the player who will see the message (-1 for all players)

*time* – time for which the message will be shown

---

---

## SiegeTown

SiegeTown – initiate the siege of the town

## Syntax

**SiegeTown**(heroName, townName, arenaName = "");

## Description

This function initiates the siege of town *townName* by the hero *heroName*; either the town's on-map name or the reference to this town's description in the resources base can be specified as the town's name. In both cases, the combat will be run just as a usual on-map town siege, that is – with the use of information like the town's garrison, the hero positioned in it, the town's combat script, etc. The *arenaName* parameter allows to replace the standard combat arena with any other.

*heroName* – the name of the hero for whom the combat is to be started

*townName* – the name of the town, which can be either the town's on-map name or the reference to this town's description in the resources base.

*arenaName* – a reference to the resource of the arena on which the combat is to take place ("" by default, which means fighting at the town's standard arena)

---

---

## StartCombat

StartCombat – start a combat with the specified parameters

## Syntax

```
StartCombat(heroName
             enemyHeroName
             enemyHeroName
             creaturesCount
             creatureType[1]
             creatureAmount[1]
             ...
             creatureType[Count]
             creatureAmount[Count]
             combatScriptName
             combatFinishTrigger
             arenaName
             allowQuickCombat
             );
```

## Description

Starts combat for the hero *heroName*, against the specified set of creatures and hero (if such a hero is given). The enemy hero's existing army is ignored. It is possible to specify a script that will be run in the beginning of the combat. Also, it is possible to specify a function that will be invoked in the end of the combat and get name of the hero for whom the combat has been started and the combat's results as its arguments.

*heroName* – the name of the hero for whom the combat is to be started

*enemyHeroName* – the name of the enemy hero (*nil* – for none, fighting against creatures only)

*creaturesCount* – number of creatures to be fought

The *creaturesCount* must follow after the ‘creatures count’ parameter:

*creatureType* – creatures’ type

*creatureAmount* – number of creatures of this type

*combatScriptName* – name of the script to be run in the beginning of the combat (*nil* by default, which means running no scripts)

*combatFinishTrigger* – name of the function (map script) that will be run in the end of the combat. This function must receive two parameters: the name of the fighting hero and the result of the combat (*nil* – if the hero has lost and not *nil* – if the hero has won).

*arenaName* – a reference to the resource of the arena on which the combat is to take place (*nil* by default, which means that the arena will be determined according to the terrain type on which the hero is standing)

*allowQuickCombat* – if not *nil*, the combat will be run in the Quick Combat mode in case if that mode is on in the game's options and the hero *heroName* does not have Quick Combat option turned off (*nil* by default, which means running a usual combat)

---

---

## StartCutScene

StartCutScene – run a cutscene

### Syntax

```
StartDialogScene(cutSceneName, callback = "", saveName = "");
```

### Description

This function runs the specified cutscene.

*cutSceneName* – the reference to the cutscene in the resources base

*callback* – name of the function to be invoked after the cutscene is played

*saveName* – name of the save file (see the *Save* function) to be created before the cutscene

---

---

# StartDialogScene

StartDialogScene – run a dialogue scene

## Syntax

```
StartDialogScene(dialogSceneName, callback = "", saveName = "");
```

## Description

This function runs the specified dialogue scene.

*dialogSceneName* – the reference to the cutscene in the resources base

*callback* – name of the function to be invoked after the dialogue scene is played

*saveName* – name of the save file (see the *Save* function) to be created before the dialogue scene

### Example 1. Example:

```
StartDialogScene("/DialogScenes/C6/M4/C1/DialogScene.xdb#xpointer(/DialogScene)")
```

---

---

# StopPlaySound

StopPlaySound – stop playing the cycled sound

## Syntax

```
StopPlaySound(loopingSoundID);
```

## Description

This function stops playing the specified cycled sound.

*loopingSoundID* – the sound ID, as returned by the functions *Play2DSound* / *Play3DSound* for cycled sounds

---

---

# TeachHeroSpell

TeachHeroSpell – gives the specified spell to the hero

## Syntax

**TeachHeroSpell**(heroName, spell);

## Description

This function tries to give the spell to the hero. If the hero already knows that spell, the function returns `nil`, otherwise it returns `not nil`.

*heroName* – the hero's name

*spell* – the spell ID

---

---

## TransformTown

TransformTown – change the type of the on-map town

### Syntax

**TransformTown**(townName, type);

### Description

This function changes the type of the on-map town to the what is specified. (In the current realization, all information regarding the town, except its name and number of the player who owns it, is lost).

*townName* – town name

*type* – new type of the town; can take on one of the following values:

TOWN\_HEAVEN,

TOWN\_PRESERVE,

TOWN\_ACEDEMY,

TOWN\_DUNGEON,

TOWN\_NECROMANCY,

TOWN\_INFERNO.

---

---

## Trigger

Trigger – set on/off the handler functions that work with events in the world

## Syntax

**Trigger**(triggerType, ..., functionName);

## Description

If the *functionName* parameter is not equal to `nil`, the function sets the handler function with that name for the specified in-game event. Otherwise the function removes the handler from this event. For each event, there can be only one handler function set. Setting another function onto the event which is already handled will remove the first handler.

*triggerType* – in-game event type; can take on one of the following values:

NEW\_DAY\_TRIGGER – new day begins

WAR\_FOG\_ENTER\_TRIGGER – the hero enters the fog of war (the behavior of the fog of war is controlled by the *SetWarfogBehaviour* function)

PLAYER\_ADD\_HERO\_TRIGGER – the player has a new hero

PLAYER\_REMOVE\_HERO\_TRIGGER – the player has lost a hero

OBJECTIVE\_STATE\_CHANGE\_TRIGGER – the objective's status has been changed

OBJECT\_CAPTURE\_TRIGGER – the object has been captured

OBJECT\_TOUCH\_TRIGGER – the interactive object has been interacted with

TOWN\_HERO\_DEPLOY\_TRIGGER – the hero leaves the town

REGION\_ENTER\_AND\_STOP\_TRIGGER – the hero enters the region (and must stay within it)

REGION\_ENTER\_WITHOUT\_STOP\_TRIGGER – the hero enters the region (and does not have to stay within it)

HERO\_LEVELUP\_TRIGGER – the hero gains a level

One or more parameters can follow the even type, determining, the events of which of the in-game objects have to be handled. The number and meaning of these parameters depend on the even type:

for NEW\_DAY\_TRIGGER – none

for PLAYER\_ADD\_HERO\_TRIGGER and PLAYER\_REMOVE\_HERO\_TRIGGER – the player's ID

for OBJECTIVE\_STATE\_CHANGE\_TRIGGER – the objective's name

for OBJECT\_CAPTURE\_TRIGGER and OBJECT\_TOUCH\_TRIGGER – the on-map object's name

for TOWN\_HERO\_DEPLOY\_TRIGGER – the town's name

for REGION\_ENTER\_AND\_STOP\_TRIGGER and REGION\_ENTER\_WITHOUT\_STOP\_TRIGGER – the region's name

for HERO\_LEVELUP\_TRIGGER – the hero's name

*functionName* – name of the handler function

When the trigger works, the following parameters are sent to the handler function:

NEW\_DAY\_TRIGGER – none

WAR\_FOG\_ENTER\_TRIGGER – the hero's name

PLAYER\_ADD\_HERO\_TRIGGER – the hero's name

PLAYER\_REMOVE\_HERO\_TRIGGER – the hero's name and the name of the hero who defeated him or her (nil – if the hero has been fired)

OBJECTIVE\_STATE\_CHANGE\_TRIGGER – the ID of the player whose objective has changed its status

OBJECT\_CAPTURE\_TRIGGER – the old owner of the object, the new owner of the object, the name of the hero who captured it (if any), and the name of the object itself

OBJECT\_TOUCH\_TRIGGER – the hero's name and the object's name

TOWN\_HERO\_DEPLOY\_TRIGGER – the hero's name

REGION\_ENTER\_AND\_STOP\_TRIGGER и REGION\_ENTER\_WITHOUT\_STOP\_TRIGGER – the hero's name

HERO\_LEVELUP\_TRIGGER – none

---

---

## UnblockGame

UnblockGame – unblock the user interface and the AI work

### Syntax

```
UnblockGame(void);
```

### Description

The command unblocks the user interface, the AI work, and as the camera management that have been blocked by the *BlockGame* command.

---

---

# UnreserveHero

UnreserveHero – make the hero who has been reserved after a certain player available for all other players again

## Syntax

**UnreserveHero**(heroName);

## Description

The hero stops being reserved after a certain player and takes part in the heroes' assignment in the players' taverns in the beginning of every new week.

NB: If the is alive when the function is invoked, he or she stops being reserved but is not removed from the list of heroes controlled by the player. Such a hero can only become available for other players by the same ways as other heroes who belong to someone (that is: if he or she dies, flees the battlefield and is not hired again till the end of the week, is fired, etc.). But if the hero is dead when the function is invoked, he or she becomes immediately available for all players except the one who owned him or her before, because he or she is considered as one who lost a combat.

*heroName* – name of hero reserved after a player.

---

---

# Win

Win – the human player wins the game when the function is invoked

## Syntax

**Win**(void);

## Description

This the function should only be used in the single-player mode. It generates an error if invoked in the multiplayer mode. When the function is invoked, the human player wins immediately, and all the AI players lose.

---

---

# COMBAT

---

---

## Prepare

Prepare – preparation for the combat

### Syntax

```
Prepare(void);
```

### Description

This function is invoked before the players begin arranging their troops at the battlefield. Troops arrangement will only begin after this function finishes its work.

---

---

## Start

Start – the start of the combat

### Syntax

```
Start(void);
```

### Description

This function is invoked after the players have arranged their troops at the battlefield and immediately before the combat begins. The combat will only begin after this function finishes its work.

---

---

## IsHuman

IsHuman – determine whether there is a human playing for the certain party

### Syntax

```
IsHuman(side);
```

### Description

This function returns not `nil` if the specified party that takes part in combat belongs to a human (regardless of whether this human controls the party's actions, or they are under automatic control), and `nil` if not.

*side* – the number of the party; can be `ATTACKER` or `DEFENDER`.

---

---

## IsComputer

IsComputer – determine whether there is the AI playing for the certain party

## Syntax

**IsComputer** (side);

## Description

This function returns not `nil` if the specified party that takes part in combat belongs to the AI, and `nil` if not.

*side* – the number of the party; can be `ATTACKER` or `DEFENDER`.

---

---

## SetControlMode

SetControlMode(side, mode) – set the combat control mode for the human player

## Syntax

## Description

The command turns on/off the automatic combat mode and sets on/off the human player's ability to switch this mode manually. The specified party must be human-controlled.

*side* – the number of the party; can be `ATTACKER` or `DEFENDER`.

*mode* – number of the mode; one of the following actions is performed according to it:

`MODE_NORMAL` – the status of the automatic combat mode is not changed, and its blocking is set off.

`MODE_MANUAL` – the automatic combat mode is turned off, and the player's ability to switch it is set off.

`MODE_AUTO` – the automatic combat mode is turned on, and the player's ability to switch it is set off.

---

---

## EnableAutoFinish

EnableAutoFinish – turn on/off the standard mechanism of checking the combat's results

## Syntax

```
EnableAutoFinish(enable);
```

## Description

The parameter sent to the command determines whether the combat will be finished if one of the parties has no more troops. If parameter is equal to `nil`, the combat will not be finished. This mode is on by default.

---

---

## Finish

Finish – finish the combat

## Syntax

```
Finish(winnerSide);
```

## Description

The command finishes the combat, and the party *winnerSide* is considered as winner.

*winnerSide* – the party that wins the combat, can be `ATTACKER` or `DEFENDER`.

---

---

## GetAttackerHero

GetAttackerHero – return the attacking party's hero

## Syntax

```
GetAttackerHero(void);
```

## Description

This function returns the name of the attacking party's hero, or `nil` if there is no hero.

---

---

## GetAttackerCreatures

GetAttackerCreatures – return the attacking party's creatures

## Syntax

```
GetAttackerCreatures(void);
```

## Description

This function returns an array of names of the attacking party's creatures.

---

---

## GetAttackerWarMachines

GetAttackerWarMachines – return the attacking party's war machines

### Syntax

```
GetAttackerWarMachines(void);
```

### Description

This function returns an array of names of the attacking party's war machines.

---

---

## GetAttackerWarMachine

GetAttackerWarMachine – return the attacking party's war machine of the specified type

### Syntax

```
GetAttackerWarMachine(type);
```

### Description

This function returns the name of the attacking party's war machine of the specified type, or `nil` if it does not exist.

*type* – the type of the war machine; can take on one of the following values:

WAR\_MACHINE\_BALLISTA – ballista

WAR\_MACHINE\_CATAPULT – catapult

WAR\_MACHINE\_FIRST\_AID\_TENT – first aid tent

WAR\_MACHINE\_AMMO\_CART – ammo cart

---

---

## GetDefenderHero

GetDefenderHero – return the defending party's hero

### Syntax

```
GetDefenderHero(void);
```

### Description

This function returns the name of the defending party's hero, or `nil` if there is no hero.

---

---

## GetDefenderCreatures

GetDefenderCreatures – return the defending party's creatures

### Syntax

```
GetDefenderCreatures(void);
```

### Description

This function returns an array of names of the defending party's creatures.

---

---

## GetDefenderWarMachines

GetDefenderWarMachines – return the defending party's war machines

### Syntax

```
GetDefenderWarMachines(void);
```

### Description

This function returns an array of names of the defending party's war machines.

---

---

## GetDefenderWarMachine

GetDefenderWarMachine – return the defending party's war machine of the specified type

## Syntax

```
GetDefenderWarMachine(type);
```

## Description

This function returns the name of the defending party's war machine of the specified type, or `nil` if it does not exist.

*type* – the type of the war machine; can take on one of the following values:

`WAR_MACHINE_BALLISTA` – ballista

`WAR_MACHINE_CATAPULT` – catapult

`WAR_MACHINE_FIRST_AID_TENT` – first aid tent

`WAR_MACHINE_AMMO_CART` – ammo cart

---

---

## GetDefenderBuildings

`GetDefenderBuildings` – return the defending party's buildings

## Syntax

```
GetDefenderBuildings(void);
```

## Description

This function returns an array of names of the defending party's buildings.

---

---

## GetDefenderBuilding

`GetDefenderBuilding` – return the defending party's building of the specified type

## Syntax

```
GetDefenderBuilding(type);
```

## Description

This function returns the name of the defending party's building of the specified type, or `nil` if it does not exist.

*type* – building type, can take on one of the following values:

BUILDING\_WALL – wall

BUILDING\_GATE – gate

BUILDING\_LEFT\_TOWER – left tower

BUILDING\_BIG\_TOWER – central tower

BUILDING\_RIGHT\_TOWER – right tower

BUILDING\_MOAT – moat

---

---

## IsAttacker

IsAttacker – determine whether the object belongs to the attacking party

### Syntax

**IsAttacker**(unitName);

### Description

This function returns not `nil` if the object belongs to the attacking party, and `nil` if not.

*unitName* – the object's name

---

---

## IsDefender

IsDefender – determine whether the object belongs to the defending party

### Syntax

**IsDefender**(unitName);

### Description

This function returns not `nil` if the object belongs to the defending party, and `nil` if not.

*unitName* – the object's name

---

---

## IsHero

IsHero – determine whether the object is a hero

### Syntax

```
IsHero(unitName);
```

### Description

This function returns not `nil` if the object is a hero, and `nil` if not.

*unitName* – the object's name

---

## IsCreature

IsCreature – determine whether the object is a creature

### Syntax

```
IsCreature(unitName);
```

### Description

This function returns not `nil` if the object is a creature, and `nil` if not.

*unitName* – the object's name

---

## IsWarMachine

IsWarMachine – determine whether the object is a war machine

### Syntax

```
IsWarMachine(unitName);
```

### Description

This function returns not `nil` if the object is a war machine, and `nil` if not.

*unitName* – the object's name

---

---

## IsBuilding

IsBuilding – determine whether the object is a building

### Syntax

```
IsBuilding(unitName);
```

### Description

This function returns not `nil` if the object is a building, and `nil` if not.

*unitName* – the object's name

---

---

## GetHeroName

GetHeroName – return the hero's name

### Syntax

```
GetHeroName(unitName);
```

### Description

This function returns the hero's real name (not the battle name).

*unitName* – the object's name

---

---

## GetCreatureType

GetCreatureType – return the creature's type

### Syntax

```
GetCreatureType(unitName);
```

### Description

This function returns the creature's type.

*unitName* – the object's name

---

---

## GetCreatureNumber

GetCreatureNumber – return the number of creatures in the group

### Syntax

```
GetCreatureNumber (unitName);
```

### Description

This function returns the number of creatures in the group.

*unitName* – the object's name

---

---

## GetWarMachineType

GetWarMachineType – return the type of the war machine

### Syntax

```
GetWarMachineType (unitName);
```

### Description

This function returns the type of the war machine. It can return one of the following values:

WAR\_MACHINE\_BALLISTA – ballista

WAR\_MACHINE\_CATAPULT – catapult

WAR\_MACHINE\_FIRST\_AID\_TENT – first aid tent

WAR\_MACHINE\_AMMO\_CART – ammo cart

*unitName* – the object's name

---

---

## GetBuildingType

GetBuildingType – return the building's type

## Syntax

```
GetBuildingType(unitName);
```

## Description

This function returns the building's type. It can return one of the following values:

BUILDING\_WALL – wall

BUILDING\_GATE – gate

BUILDING\_LEFT\_TOWER – left tower

BUILDING\_BIG\_TOWER – central tower

BUILDING\_RIGHT\_TOWER – right tower

BUILDING\_MOAT – moat

*unitName* – the object's name

---

---

## GetUnitPosition

GetUnitPosition – return the object's coordinates

### Syntax

```
GetUnitPosition(unitName);
```

### Description

This function returns the x and y coordinates of the object's location.

*unitName* – the object's name

---

---

## AddCreature

AddCreature – add creatures of the specified type to one of the parties

### Syntax

```
AddCreature(side, type, number, x = -1, y = -1);
```

## Description

This function adds *number* of creatures of the *type* to the party *side*. If the *x* and *y* parameters are set, the creature appears in the free tile closest to (*x*, *y*); otherwise it appears in the random location at the arena. It is also possible to set only one of the tile's coordinates. Creatures that are added by this command remain in the hero's army after the combat (if there is room for them there).

*side* – the party to which the creature is summoned; can take on one of the two values:

ATTACKER – the attacking party

DEFENDER – the defending party

*type* – type of the creature

*number* – number of the creatures

*x*, *y* – coordinates of the tile in which the creature must appear (-1 is for random coordinates)

---

---

## EnableCinematicCamera

EnableCinematicCamera – turn the cinematic camera on/off in the combat

### Syntax

**EnableCinematicCamera**(enable);

### Description

This function works correctly only for the current combat. The camera can now switch to cinematic mode only if this is allowed in the settings and not prohibited by the script.

---

---

## TUTORIAL

---

---

### IsTutorialItemEnabled

IsTutorialItemEnabled – get to know whether the tutorial element will be shown

### Syntax

**IsTutorialItemEnabled**(name);

## Description

*name* – name of the tutorial element

---

---

## IsTutorialMessageBoxOpen

IsTutorialMessageBoxOpen – get to know whether any window with the tutorial text is currently open

## Syntax

**IsTutorialMessageBoxOpen**();

## Description

This function returns true if there is a window invoked by the TutorialMessageBox function present at the screen.

---

---

## TutorialActivateHint

TutorialActivateHint – set on the trigger for a specified window's appearance, and a window with the tutorial text is shown when the trigger works

## Syntax

**TutorialActivateHint**(stringID);

## Description

This function finds the descriptor element in UIConsts.tutorialOptions.hints by its *stringID*. When the window specified in the descriptor shows again, another window with the text (similar to that invoked by the TutorialMessageBox function) will be shown.

---

---

## TutorialMessageBox

TutorialMessageBox – show the window with the tutorial text

## Syntax

```
TutorialMessageBox(stringID);
```

## Description

This function finds the descriptor element in `UIConsts.tutorialOptions.hints` by its *stringID* and shows a window with the text to which that element is referring.

---

---

## TutorialSetBlink

TutorialSetBlink – turn the control element highlighting on/off

### Syntax

```
TutorialSetBlink(stringID, turnOn);
```

### Description

This function finds the descriptor element in `UIConsts.tutorialOptions.hints` by its *stringID* and turns the highlighting on/off (depending on the *turnOn* values).

---

---

## TOWN

---

---

## HeroHired

HeroHired – a hero is hired

### Syntax

```
HeroHired(name);
```

### Description

This function is invoked if a player hires the hero.

---

---

## CreatureHired

CreatureHired – a creature is hired

### Syntax

`CreatureHired(type, number);`

### **Description**

This function is invoked if a player hires the troops.

---

---