

Universe Script

Table of contents

Introduction	cam.setSmoothStop	game.ping
Hello World	cam.setSpeed	game.playMusic
Comments	Enemy	game.resetEvents
Variables	enemy.assignToSpawner	game.resumeTime
Var Objects as Numbers, Strings and Booleans	enemy.getPos	game.scheduleCheckpointLoad
Control Statements	enemy.isMoving	game.scheduleCheckpointSave
Waiting	enemy.isSpawning	game.setBars
Logging	enemy.kill	game.setCompletedBeforeFor Testing
Restricted Keywords	enemy.moveTo	game.setEvECollisionEvents
Var Objects as Arrays	enemy.setAI	game.setSkipable
Var Objects as Code	enemy.setInvincibility	game.setTeamCount
Bot	enemy.setPos	game.setTime
bot.getPos	enemy.setVariable	game.showScore
bot.isMoving	enemy.setVisibility	game.showTime
bot.isSpeaking	Game	game.usePvEScore
bot.lookAtWorld	game.blockPause	game.usePvPScore
bot.moveToScreen	game.end	game.win
bot.moveToWorld	game.ended	Global
bot.say	game.fadeIn	and / &&
bot.setAutoLook	game.fadeOut	asCode
bot.setBoxWidth	game.getCompletedBefore	asScore
bot.setColor	game.getEvent	asString
bot.setColorS	game.getEventValue	asTime
bot.setEmotion	game.getLocalPlayer	create
bot.setHair	game.getMinimumScore	delete
bot.setLookSpeed	game.getPlayer	endScript
bot.setName	game.getPlayerCount	exists
bot.setPosScreen	game.getScoreLimit	exit
bot.setPosWorld	game.getSkip	if
Camera	game.getTeam	import
cam.follow	game.getTeamCount	not / !
cam.getPos	game.getTime	or /
cam.getPosZ	game.getTimeLimit	setLogging
cam.isMoving	game.isFading	startThread
cam.isZooming	game.isFFA	wait
cam.moveTo	game.killAllEnemies	while
cam.moveToZ	game.killEnemiesOfSpawner	xor
cam.setPos	game.lose	+
cam.setPosZ	game.message	-
	game.pauseTime	

*	player.getAverageDronePos	playerSpawn.setState
>	player.getInput	playerSpawn.setTeam
<	player.getLevel	Screen
>=	player.getMultiplier	screen.print
<=	player.getPos	Script Position
==	player.getScore	scriptPosition.getPos
!=	player.getSteamName	Team
Logic	player.getTeam	team.addScoreBonus
logic.get	player.hasMarker	team.getColor
logic.set	player.isAlive	team.getLobbySlotCount
logic.trigger	player.isDashing	team.getName
Math	player.isMoving	team.getPlayer
math.abs	player.isShielded	team.getPlayerCount
math.atan	player.isShooting	team.getScore
math.atan2	player.isSwinging	team.getScoreBonus
math.ceil	player.isTeleporting	team.setScoreBonus
math.cos	player.kill	Variables
math.floor	player.moveTo	variable.asCode
math.length	player.removeAllMarkers	variable.asScore
math.mod	player.removeMarker	variable.asString
math.pi	player.setDeathHandling	variable.asTime
math.pow	player.setDash	variable.get
math.rnd	player.setDroneLimit	variable.getSize
math.sin	player.setInvincibility	variable.insert
math.sqrt	player.setLevel	variable.remove
math.tan	player.setPlayerControl	variable.set
math.toDeg	player.setPos	variable.setAt
math.toRad	player.setScore	variable.+=
math.trunc	player.setShoot	variable.-=
Player	player.setSmoothStop	variable.*=
player.addMarker	player.setTeam	variable.=/
player.addScore	player.shoot	Event Handling
player.allowDroneInactivity	player.spawnSwarm	collision
player.dash	player.teleportTo	enemyEvent
player.damage	Player Spawn	

Introduction

In this guide you will learn how to use Universe Script (US) in your levels to realize functionality that is not offered by the level editor itself. Scripts can be simple one-liners that make a bot say something, up to complex sets of scripts that implement new game modes.

This chapter uses [and] to quote source code where appropriate to avoid confusion with quotation marks that are syntactically relevant.

Hello World

Universe Script is built around objects that supply functions. Functions support a flexible amount of parameters and return values. There are no fixed types. A fully functional script may look something like this:

```
bot.say( "Hello World!" );
```

This line assumes that there already is an object called [bot]. It supplies the function [say]. This function is called with the string ["Hello World!"] as a parameter. It returns nothing of interest (technically it returns [ok]). The statement is concluded by a semicolon. Nearly everything in US uses this structure. So at least for now we'll consider this the only valid syntax. Return values can be used in nested calls, but still, the same structure applies.

While this resembles C++ or Java syntax, they are completely different languages. This will become more obvious later. One of many differences is that no new object types can be written in US, you can only use the ones supplied by the game.

Variables, the bot we just used, the camera, enemies, the player, all of these are objects. Some objects, like the camera or the player, are created by the game and therefore can be used directly. Other objects, like bots, additional enemies or variables must be created before they can be used. Objects can be deleted via script (typically when you want to stop using them) but don't have to be. They will automatically be deleted when their scope ends, but more about that later.

Let's look at how the object we used ([bot]) could have been created:

```
create( "Bot", "bot" );
```

You may ask why there is no object that supplies the function [create]. This is for your convenience. Technically this line is equivalent to:

```
se.create( "Bot", "bot" );
```

[se] stands for "Script Engine" and all functions supplied by the script engine itself can be accessed globally. Note that this means you cannot name your own objects [se].

["Bot"] is a string containing the name of the object type that will be created and ["bot"] is the name this specific object will be given. Now that the object [bot] exists, it can be used. It supplies the functions unique to the object type [Bot] and has its own unique properties like position or color.

The complete "Hello World" script looks like this:

```
create( "Bot", "bot" );  
bot.say( "Hello World!" );
```

Comments

SU supports comments in the form of line comments:

```
// this line will not be executed  
create( "Bot", "bot" ); // creates a bot and ignores this comment
```

Variables

You may be used to something like this:

```
a = b + c;
```

Lines like this one will not work due to the simplistic nature of US. In US, variables are objects of the type [Var]. Assuming the variables have already been created, the line above would look like this:

```
a.set( +( b.get(), c.get() ) );
```

What you see here is the Var object's functions [get] and [set] in action. You cannot write [b] and expect it to represent its value. The same "object.function(parameters)" structure must be applied to access it. [b.get] will return b's value. [a.set] will store the parameter as a's new value.

And then there is [+(x, y)]. This, like [create], is a function supplied by the script engine itself and is therefore equivalent to [se.+(x, y)]. It returns the result of adding x and y.

The code above might be hard to take in at first, but look at it again, does it make more sense now? It's a bit hard to read but how it works is actually quite simple. A language definition for Notepad++ will be available soon, it gets much easier with source highlighting.

Please note that it is not necessary to do math or all the other complicated stuff coming up next to do many fun things in US, like scripting a non-interactive scene. And often the level editor's features can be used to "connect" scripts to create interactivity.

Var Objects as Numbers, Strings and Booleans

Numbers, handled via 4 Byte Floats, and Booleans don't have quotation marks, but strings do. Var objects accept whatever type they receive.

```
a.set( 3 );  
a.set( -3 );  
a.set( 3.3 );  
a.set( "Three" );  
a.set( true );  
a.set( false );
```

A few more examples of how to work with these data types:

```
a.set( and( false, true ) );    // sets a to false  
a.set( ==( 3, 3 ) );           // sets a to true  
a.set( not( false ) );         // sets a to true  
a.set( !( false ) );           // another way of using [not]  
  
a.set( +( 3, 5 ) );            // sets a to 8  
a.set( +( "Three", " Five" ) ); // sets a to "Three Five"
```


Control Statements

The available control statements are [if] and [while]. [if] looks like this:

```
if( condition )  
    // will be executed if condition is true  
else  
    // will be executed if condition is false  
end
```

Note that [else] is not mandatory and that [if], [else] and [end] are not followed by semicolons.

[while] works the same way and looks like this:

```
while( condition )  
    // will be executed as long as condition is true  
end
```

A classic for-loop can be created using [while] and a Var object:

```
create( "Var", "i", 0 ); // Var objects can be initialized like this  
while( <( i.get(), 10 ) )  
    i.+=( 1 );  
end  
delete( "i" ); // objects can be deleted
```

Waiting

The script engine executes scripts 60 times per second. Each execution cycle will continue until the global function [wait()] is called. There are a few functions that can wait internally, but in general a script needs to call [wait()] when appropriate to yield its further execution for now and allow the game to progress in time.

Logging

The global function [setLogging] can be used to enable logging of script execution. This is the only way of checking for errors besides watching what happens in the game. Enabling logging for long scripts (containing loops) can heavily decrease the game's performance. It is advised to only enable logging for sections you are currently working on. The log can be found in your documents folder under "\\Dedication Games\Swarm Universe\[YOUR STEAM ID]\log.txt".

```
setLogging( true );  
  
// instructions here will produce log entries  
  
setLogging( false );
```

Note that this only affects the current script and not other scripts that may be running at the same time.

Restricted Keywords

Some words are already in use by US itself and are therefore not allowed as object names. These include names like [se], [game] or [math] (object names that are already in use) as well as [.] and the names of global functions like [and], [==] and [create]. You should be fine as long as you use some common sense when naming your objects. Errors resulting from this may be hard to spot.

Var Objects as Arrays

An array is a series of strings or numbers (or other content), separated by [,]. This is important because arrays can be used to fill multiple parameters of a function at once. This is also what is returned by a function that returns multiple values.

The Var object's [get] function supports an index parameter. Its [set] function supports multiple parameters to set an array in one call. Since this would not allow for specifying an index at which to set the given value, there is another function called [setAt]. It takes an index and a single value.

```
a.setAt( 0, "Bot" );  
a.setAt( 1, "bot" );
```

This is equivalent to

```
a.set( "Bot", "bot" );
```

If we now call [a.get(0)] the function returns ["Bot"]. With index 1 it returns ["bot"]. But if we call [get] without any parameter it will return the whole array: ["Bot", "bot"]. What this means is we could now do this:

```
create( a.get() );
```

which will translate to:

```
create( "Bot", "bot" );
```

If a function returns multiple values, like [bot.getPos()] does, we could do

```
a.set( bot.getPos() );
```

Now a's value is something like [123.4, 432.1, 0], three coordinates for a 3D position. We could use this in some setPos call that would take three parameters, or we could access and modify those individually using indexes. This also means that the following pseudocode would work by passing three return values into three parameters:

```
setPos( getPos() );
```

Var Objects as Code

Arrays make use of this, you may have noticed. You can even write your own "functions" using Var objects. They are not fully qualified functions (compared to the ones supplied by objects) since they support no parameters or return values but this behavior can be achieved using other Var objects.

The key here is the Var object's function [asCode]. What it does is simple, it returns the variable's value, but without quotation marks. And a string without quotation marks is code. To give an example:

```
a.set( "b.get()" ); // "b.get()" is a string, it will not be executed
a.set( b.get() );  // b.get() is not a string, it will be executed
```

Therefore we can do this:

```
a.set( "b.get()" ); // value is now ["b.get()"]
a.asCode();
```

[a.asCode()] will return [b.get()]. Note the lack of quotation marks. If we would have used [.get()], the return value would include the string's quotation marks. Without them [a.asCode()] is replaced by [b.get()], something that can be executed and therefore will be executed. So after this additional step, the line of code gets replaced by the content of [b].

It's important to know what actually happens in order to understand how "pointers" or "aliases" or "references" can be implemented. Look at this:

```
a.set( "b" );
a.asCode().get();
```

The second line executes from left to right. [a.asCode()] gets replaced by [b], therefore what remains is [b.get()] which will be executed and replaced by the content of b. This makes use of how JS is executed. Note that the structure [object.function().function();] is technically not valid syntax as described in the beginning. But going from left to right, [a.asCode()] is executed first and replaced by its return value. That's what makes this syntax valid when the line is re-evaluated.

Here is some more complex code, something that could be called a function definition:

```
// an instruction can be distributed over multiple lines
a.set( "
    b.set( \"One\" );
    b.set( \"Two\" );
    b.set( \"Many\" );
" );
```

Time to take a breath. This is a single "line" of code. Note how everything is wrapped in ["]. [a] is simply set to a single string, distributed over multiple lines. What's very important about this is that our calls inside the string need their own ["] to state their parameters. Hence the backslash in front of the quotation marks. It protects the quotation mark from being seen as the end of the string we set [a] to. To understand this, you need to know that [asCode] not only removes the quotation marks at the beginning and the end of the contained string, it also unwraps protected quotation marks. Now if we call [a.asCode();] this will translate to:

```
b.set( "One" ); b.set( "Two" ); b.set( "Many" );
```

What happens next is that the script engine detects the semicolons. They tell it that these are individual lines. So it executes the first one and enqueues the other two.

It's important to know that semicolons do this in order to understand why you cannot call such a user-made function as a parameter of another function. The line you would use it in would be separated at the first semicolon and become invalid. Also there is no return value that represents the function in its entirety.

What you could do is call your function a line before the result is needed in a non-nested way. The function could fill another Var with its result and that's the Var that could be used in the next line. If such a function would need parameters as well, the same concept of filling other Var's in advance can be employed. Here is an example for all that:

```
compute.set( "  
    result.set( parameter.get() );  
" );  
  
parameter.set( "UseThis" );  
compute.asCode();  
result.get(); // will return ["UseThis"]
```

You may have noticed the tedium of protecting quotation marks inside functions with backslashes. To make this easier, there is an alternative form of quotation marks available that supports a start and an end (like brackets). This will translate to protected quotation marks as seen before:

```
a.set( \  
    b.set( "One" );  
    b.set( "Two" );  
    b.set( "Many" );  
\  
);
```

Note that there is also a global function [asCode] available. There are cases in which the code you wish to execute is not contained in a Var object. Therefore this:

```
compute.asCode();
```

is equivalent to this:

```
asCode( compute.get() );
```

Also note that this is mostly used for basic things like referencing variables. If you are not actually in need of functions that are written during runtime, using the macro command [import] is often the better choice for function-like behavior. [import] will simply paste the content of another script before running your code.

Bot

bot.getPos

Returns the bot's current world position.

Syntax

```
x, y, z bot.getPos ( [shiftX, shiftY, shiftZ] );
```

Parameters

shiftX, shiftY, shiftZ

Numerical coordinates that will be added to `x`, `y`, `z` to get a position relative to the bot, for example, a position to its right.

Return Values

x, y, z

Numerical coordinates that are always the bot's world coordinates, even if it has been moved to the screen plane.

Examples

```
// cause the bot to look to its left
bot.lookAtWorld( bot.getPos( -10, 0, 0 ) );
```

bot.isMoving

Returns if the bot is currently moving.

Syntax

```
moving bot.isMoving ();
```

Return Values

`moving`

Boolean that is only true if the bot is currently moving.

Examples

```
// wait until the bot stops moving  
wait( bot.isMoving() );
```

bot.isSpeaking

Returns if the bot is currently speaking.

Speaking means that the bot's message box is currently shown.

Syntax

```
speaking bot.isSpeaking ();
```

Return Values

`speaking`

Boolean that is only true if the bot is currently speaking.

Examples

```
// wait until the bot stops speaking  
wait( bot.isSpeaking() );
```

bot.lookAtWorld

Causes the bot to look at a given point in the world.

By default the bot looks at the player's eyes and into certain random directions (see [bot.setAutoLook](#)). The only exceptions are when the bot is moving towards a world target (see [bot.moveToWorld](#)), in which case the bot looks forward, or when it is forced to look at a certain point in the world (see [bot.lookAtWorld](#)).

Syntax

```
bot.lookAtWorld ( x, y, z );  
bot.lookAtWorld ( false );
```

Parameters

`x, y, z`

Numerical world coordinates for the bot to look at.

`false`

Causes the bot to use its default look behavior.

Examples

```
// cause the bot to look at player1's current swarm position and then wait for 3 seconds  
bot.lookAtWorld( player1.getPos(), 0 );  
wait( 180 );  
  
// cause the bot to activate its default look behavior again  
bot.lookAtWorld( false );
```

bot.moveToScreen

Causes the bot to move to a given point on the screen plane.

As long as the bot is on the screen plane, it will automatically match the camera's movement to always stay on the same screen position. Additionally, it will always look at the player's eyes and into certain random directions (see [bot.setAutoLook](#)) unless it is forced to look at a specific point in the world (see [bot.lookAtWorld](#)).

Syntax

```
bot.moveToScreen ( x, y, z [, "wait" ] );
```

Parameters

x, *y*, *z*

Numerical screen coordinates for the bot to move to. The *z* coordinate can be used to move the bot closer to or farther away from the camera. The higher *z* is, the further away the bot will move from the camera.

"wait"

The script will wait until the bot has reached its *x*, *y*, *z* target.

Examples

```
// move the bot to the screen plane, above the screen center
bot.moveToScreen( 0, 100, 0, "wait" );

// move closer
bot.moveToScreen( 0, 100, -25, "wait" );
```

bot.moveToWorld

Causes the bot to move to a given point in the world.

Unless the bot has been forced to look at a specific point (see [bot.lookAtWorld](#)), it will look forward while moving towards a world target or look at the player's eyes and into certain random directions (see [bot.setAutoLook](#)) while not moving.

Syntax

```
bot.moveToWorld ( x, y, z [, "wait" ] );
```

Parameters

x, *y*, *z*

Numerical world coordinates for the bot to move to.

"wait"

The script will wait until the bot has reached its *x*, *y*, *z* target.

Examples

```
// move the bot to the left of the player's swarm's current position
bot.moveToWorld( player1.getPos( -50, 0 ), -200, "wait" );
```

bot.say

Causes the bot to say a given message.

Syntax

```
bot.say ( [message] [, "wait"] );
```

Parameters

message

String containing the message. The message may contain emotion names (see [bot.setEmotion](#)) encased in brackets. The bot will switch its emotion as soon as it reaches the specific part of the message. Quotation marks and brackets with a leading backslash will be part of the bot's message. If no `message` is passed, the bot will stop speaking immediately.

"wait"

The script will wait until the bot has finished speaking.

Examples

```
// cause the bot to say "Welcome, great to see you!" and switch its emotion to "happy" once it
reaches the word "great"
bot.say( "Welcome, (happy)great to see you!", "wait" );

// show quotation marks and brackets within the bot's message
bot.say( "\\(Your text: \"Great to see you too!\\)\", \"wait\" );
```

bot.setAutoLook

Activates or deactivates the bot's auto look behavior for when it is located on the screen plane (see [bot.moveToScreen](#)).

If the behavior is active, the bot will not only toggle between the user's eye but also look into certain random directions while it is located on the screen plane. The frequency and direction of those looks are based on the bot's current emotion (see [bot.setEmotion](#)). If the bot is forced to look at a certain point in the world (see [bot.lookAtWorld](#)), the auto look behavior is not executed. The auto look behavior is on by default.

Syntax

```
bot.setAutoLook ( look );
```

Parameters

`look`

Boolean to activate (true) or deactivate (false) the bot's auto look behavior.

Examples

```
// deactivate the bot's auto look behavior
bot.setAutoLook( false );
```


bot.setBoxWidth

Sets the bot's box width.

The box is shown while the bot is saying something (see [bot.say](#)).

Syntax

```
bot.setBoxWidth ( width );
```

Parameters

width

Numerical width ranging from 300 (default) to 900.

Examples

```
// increase the bot's box width  
bot.setBoxWidth( 500 );
```

bot.setColor

Sets the bot's colors.

Syntax

```
bot.setColor ( primR, primG, primB [, secR, secG, secB] );
```

Parameters

primR, *primG*, *primB*

Numerical RGB values for the bot's primary color, each ranging from 0.0 to 1.0. The bot's primary color is used for its eye and name (see [bot.setName](#)).

secR, *secG*, *secB*

Numerical RGB values for the bot's secondary color, each ranging from 0.0 to 1.0. The bot's secondary color is used for its body, hair (see [bot.setHair](#)) and message box (see [bot.say](#)).

Examples

```
// set the primary color to cyan and the secondary color to bright pink
bot.setColor( 0, 0.7, 0.7, 1, 0, 1 );
```

bot.setColorS

Sets the bot's secondary color.

Syntax

```
bot.setColorS ( secR, secG, secB );
```

Parameters

secR, *secG*, *secB*

Numerical RGB values for the bot's secondary color, each ranging from 0.0 to 1.0. The bot's secondary color is used for its body, hair (see [bot.setHair](#)) and message box (see [bot.say](#)).

Examples

```
// set the bot's secondary color to a bright green  
bot.setColorS( 0.25, 1, 0.25 );
```

bot.setEmotion

Sets the bot's emotion.

Emotions control the bot's overall look, its blink frequency and the way the bot looks at the player's eyes and into certain random directions (see [bot.setAutoLook](#)).

Syntax

```
bot.setEmotion ( emotion );
```

Parameters

`emotion`

String containing one of the following options: "neutral", "angry", "scared", "excited", "happy", "crazy", "sleepy", "skeptical", "surprised", "bored" or "random";

Examples

```
// let the bot realize that the player is here
bot.setEmotion( "surprised" );
wait( 25 );
bot.setEmotion( "happy" );
```

bot.setHair

Sets the bot's hair style.

Syntax

```
bot.setHair ( style );
```

Parameters

style

Either false or a string containing one of the following options: "short", "long", "goatee". If *style* is false, all hair gets removed.

Examples

```
// "grow" long hair and wait 3 seconds
bot.setHair( "long" );
wait( 180 );

// "cut" the bot's hair
bot.setHair( false );
```

bot.setLookSpeed

Sets the bot's look speed controlling the rotation towards its look target.

By default the bot's look speed depends on its current emotion (see [bot.setEmotion](#)) and the distance between the bot and its current look target (see [bot.lookAtWorld](#)).

Syntax

```
bot.setLookSpeed ( speed );  
bot.setLookSpeed ( false );
```

Parameters

speed

Numerical speed value reaching from 0 (no look changes at all) to 1 (instant look changes).

false

Causes the bot to use its default look speed.

Examples

```
// cause the bot to slowly look to it's left  
bot.setLookSpeed( 0.15 );  
bot.lookAtWorld( bot.getPos( -50, 0, -5 ) );  
wait( 180 );  
  
// reactivate the bot's default look speed  
bot.setLookSpeed( false );
```

bot.setName

Sets the bot's name.

The bot's name will be displayed in the header of its message box and in the bot log. By default the bot's name is the same as its script name (see [create](#)). This function does not change the bot's script name.

Syntax

```
bot.setName ( name );
```

Parameters

`name`

String containing the bot's name.

Examples

```
// create a bot and set its name script name to "bot"
create( "Bot", "bot" );

// set the bot's name
bot.setName( "Bot-7" );

// use its unchanged script name to say something, its message box though will show "Bot-7"
bot.say( "Hello player!" );
```

bot.setPosScreen

Instantly sets the bot's position to a given point on the screen plane.

As long as the bot is on the screen plane, it will automatically match the camera's movement to always stay on the same screen position. Additionally it will always look at the player's eyes and into certain random directions (see [bot.setAutoLook](#)) unless it is forced to look at a specific point in the world (see [bot.lookAtWorld](#)).

Syntax

```
bot.setPosScreen ( x, y, z );
```

Parameters

x, *y*, *z*

Numerical screen coordinates for the bot to be moved to. The *z* coordinate can be used to move the bot closer to or farther away from the camera. The higher *z* is, the further away the bot will move from the camera.

Examples

```
// move the bot to the screen center's upper right  
bot.setPosScreen( 250, 100, 0 );
```


bot.setPosWorld

Instantly sets the bot's position to a given point in the world.

The bot will look at the player's eyes and into certain random directions (see [bot.setAutoLook](#)) unless it starts moving towards a world target (see [bot.moveToWorld](#)) or if it is forced to look at a specific point in the world (see [bot.lookAtWorld](#)).

Syntax

```
bot.setPosWorld ( x, y, z );
```

Parameters

x, *y*, *z*

Numerical world coordinates for the bot to be moved to.

Examples

```
// instantly move the bot to its left  
bot.setPosWorld( bot.getPos( -250, 0, 0 ) );
```

Camera

cam.follow

Causes the camera to follow a given script object.

Syntax

```
cam.follow ( object );
```

Parameters

object

Either false or a string containing the object's script name. If *object* is false, the camera will stop moving.

Examples

```
// follow a moving enemy called "targetEnemy" for 3 seconds
cam.follow( "targetEnemy" );
wait( 180 );

// follow the player again
cam.follow( "player1" );
```

cam.getPos

Returns the camera's current world position.

Syntax

```
x, y, z cam.getPos ( [shiftX, shiftY, shiftZ] );
```

Parameters

`shiftX`, `shiftY`, `shiftZ`

Numerical coordinates that will be added to `x`, `y`, `z`. to get a position relative to the camera, for example, a position to its right.

Return Values

`x`, `y`, `z`

Camera's current numerical world position.

Examples

```
// move the camera down  
cam.moveTo( cam.getPos( 0, -100, 0 ), "wait" );
```

cam.getPosZ

Returns the camera's current zoom level.

Syntax

```
z cam.getPosZ ( ) ;
```

Return Values

z

Camera's current numerical zoom level.

Examples

```
// zoom in  
cam.moveToZ( +( cam.getPosZ(), 100 ), "wait" );
```

cam.isMoving

Returns if the camera is currently moving.

Syntax

```
moving cam.isMoving ();
```

Return Values

`moving`

Boolean that is only true if the camera is currently moving.

Examples

```
// wait until the camera stops moving  
wait( cam.isMoving() );
```

cam.isZooming

Returns if the camera is currently zooming.

Syntax

```
zooming cam.isZooming ();
```

Return Values

zooming

Boolean that is only true if the camera is currently zooming.

Examples

```
// wait until the camera stops zooming  
wait( cam.isZooming() );
```

cam.moveTo

Causes the camera to move to a given point.

Syntax

```
cam.moveTo ( x, y [, zoom] [, "wait"] );
```

Parameters

x, y

Numerical world coordinates for the camera to move to.

zoom

Numerical level for the camera to zoom to, ranging from -25.0 (very close) to -2500.0 (very far).

"wait"

The script will wait for the camera to stop moving. If *zoom* was passed as well, the script will also wait for the camera to stop zooming.

Examples

```
// move the camera to the player's current position (but don't follow) and zoom closer  
cam.moveTo( player1.getPos(), -150, "wait" );
```


cam.moveToZ

Causes the camera to zoom to a given level.

Syntax

```
cam.moveToZ ( zoom [, "wait" ] );
```

Parameters

zoom

Either a numerical level ranging from -25.0 (very close) to -2500.0 (very far) or a string that must be "near", "normal" or "far" (which translates to a level of -350, -800 or -1100) for the camera to zoom to.

"wait"

The script will wait for the camera to stop zooming.

Examples

```
// zoom in and wait for 3 seconds
cam.moveToZ( "near", "wait" );
wait( 180 );

// zoom out
cam.moveToZ( -1500 );
```

cam.setPos

Instantly sets the camera's position to a given point in the world.

Causes the camera to stop moving.

Syntax

```
cam.setPos ( x, y [, zoom] );
```

Parameters

x, y

Numerical world coordinates for the camera to be moved to.

zoom

Numerical level for the camera to be zoomed to, ranging from -25.0 (very close) to -2500.0 (very far).

Examples

```
// instantly move the camera right (without zooming)
cam.setPos ( cam.getPos( 250, 0, 0 ) );
```

cam.setPosZ

Instantly sets the camera's zoom to a given level.

Syntax

```
cam.setPosZ ( zoom );
```

Parameters

zoom

Either a numerical level ranging from -25.0 (very close) to -2500.0 (very far) or a string that must be "near", "normal" or "far" (which translates to a level of -350, -800 or -1100) for the camera to zoom to.

Examples

```
// instantly zoom out and wait for 3 seconds
cam.setPosZ( -1750 );
wait( 180 );

// instantly reset to the normal zoom level
cam.setPosZ( "normal" );
```

cam.setSmoothStop

Activates or deactivates the camera's "Smooth Stop" handling.

If active, the handling causes the camera to make a smooth stop if it is about to reach its move target.

Syntax

```
cam.setSmoothStop ( smooth );
```

Parameters

smooth

Boolean to activate (true) or deactivate (false) the camera's "Smooth Stop" handling.

Examples

```
// activate the camera's "Smooth Stop" handling  
cam.setSmoothStop( true );
```

cam.setSpeed

Sets the camera's move and zoom speed.

Syntax

```
cam.setSpeed ( speed );
```

Parameters

speed

Either a numerical value ranging from 0.0 (no speed) to 2.0 (double speed) or a string that must be "slow", "normal" or "fast" (which translates to a speed of 0.4, 1.0 or 1.75);

Examples

```
// slowly move the camera to the player
cam.setSpeed( 0.25 );
cam.moveTo( player1.getPos(), "wait" );

// follow the player with normal speed
cam.follow( "player1" );
cam.setSpeed( "normal" );
```

Enemy

enemy.assignToSpawner

The enemy will be treated like it would have been spawned by a given spawner.

This may affect the spawner's spawn behavior and/or spawn trackers, which are connected to the given spawner.

Syntax

```
enemy.assignToSpawner ( spawner );
```

Parameters

spawner

String containing the spawner's script name.

Examples

```
// assign the enemy named "enemy1" to a spawner named "spawner3"  
enemy1.assignToSpawner( "spawner3" );
```

enemy.getPos

Returns the enemy's current world position.

Syntax

```
x, y enemy.getPos ( [shiftX, shiftY] );
```

Parameters

shiftX, shiftY

Numerical coordinates that will be added to `x`, `y` to get a position relative to the enemy, for example, a position to its right.

Return Values

x, y

Enemy's current numerical world position.

Examples

```
// move the enemy named "enemy1" to its left  
enemy1.moveTo( enemy1.getPos( -100, 0 ), "wait" );
```


enemy.isMoving

Returns if the enemy is currently moving.

Syntax

```
moving enemy.isMoving ();
```

Return Values

`moving`

Boolean that is only true if the enemy is currently moving.

Examples

```
// wait until the enemy named "enemy1" stops moving  
wait( enemy1.isMoving() );
```

enemy.isSpawning

Returns if the enemy is currently spawning.

Syntax

```
spawning enemy.isSpawning ();
```

Return Values

spawning

Boolean that is only true if the enemy is currently spawning.

Examples

```
// wait until the enemy named "enemy1" stops spawning  
wait( enemy1.isSpawning() );
```

enemy.kill

Causes the enemy to die and play its standard death animation.

The enemy will not award any points to any player. Actions that are configured to happen when the enemy is "Being Defeated" within the enemy's action management will be triggered.

Syntax

```
enemy.kill ();
```

Examples

```
// kill the enemy named "enemy1"  
enemy1.kill();
```

enemy.moveTo

Causes the enemy to move to a given point in the world.

Deactivates the enemy's AI (see [enemy.setAI](#)).

Syntax

```
enemy.moveTo ( x, y [, "wait"] );
```

Parameters

x, *y*

Numerical world coordinates for the enemy to move to.

"wait"

The script will wait until the enemy has reached its *x*, *y* target.

Examples

```
// move the enemy named "enemy1" to the current player's position  
enemy1.moveTo( player1.getPos() );
```

enemy.setAI

Activates or deactivates the enemy's AI.

Enemies with deactivated AI do not move automatically, a script needs to handle their movement. They also don't shoot.

Syntax

```
enemy.setAI ( AI );
```

Parameters

AI

Boolean to activate (true) or deactivate (false) the enemy's AI.

Examples

```
// deactivate the enemy's AI of "enemy1"  
enemy1.setAI( false );
```

enemy.setInvincibility

Activates or deactivates the enemy's script based invincibility.

This is an additional shield effect and does neither affect the enemy's spawn protection nor its shield ability.

Syntax

```
enemy.setInvincibility ( invincible );
```

Parameters

`invincible`

Either a numerical amount of ticks the enemy will be invincible for or a Boolean to activate (true) or deactivate (false) the enemy's invincibility.

Examples

```
// cause the enemy named "enemy1" to be invincible for 1 second and wait 2 seconds
enemy1.setInvincibility( 60 );
wait( 120 );

// cause it to be invincible until further notice
enemy1.setInvincibility( true );
```

enemy.setPos

Instantly sets the enemy's position to a given point in the world.

Syntax

```
enemy.setPos ( x, y );
```

Parameters

x, y

Numerical world coordinates for the enemy to be moved to. The position has to be inside of at least one field wall and outside of every obstacle wall.

Examples

```
// instantly move the enemy named "enemy1" to the player's position  
enemy1.setPos( player1.getPos() );
```

enemy.setVariable

Sets an enemy's variable to a given Boolean.

Those variables can be used by the enemy's action management (condition "Script Variable" using the local scope).

Syntax

```
enemy.setVariable ( variable, value );
```

Parameters

variable

String containing the variable's name.

value

Boolean to set the enemy's variable to.

Examples

```
// set enemy1's (an enemy) "StateA" variable, its action management might trigger something  
enemy1.setVariable( "StateA", true );
```


enemy.setVisibility

Sets the enemy's visibility.

Syntax

```
enemy.setVisibility ( visibility );
```

Parameters

visibility

Boolean to show (true) or hide (false) the enemy

Examples

```
// hide the enemy  
enemy.setVisibility( false );
```

Game

game.blockPause

Tells the game whether to block or not to block the menu's game pause.

Does not work in Multiplayer levels as there is no game pause. By default a Singleplayer level pauses when the menu is shown. The pause is therefore not blocked by default.

Syntax

```
game.blockPause ( block );
```

Parameters

block

Boolean to block (true) or not to block (false) the menu's game pause.

Examples

```
// block the menu's game pause  
game.blockPause ( true );
```

game.end

Causes the level to end.

In Singleplayer levels the player wins if the configured win condition is fulfilled, otherwise the player loses.

In Multiplayer levels the team(s)/player(s) with the highest score win(s), the rest loses.

Syntax

```
game.end ();
```

Examples

```
// end the game  
game.end();
```

game.ended

Returns if any game ending condition has been triggered.

This could be a triggered level end element, a score or time limit in Multiplayer levels or various script calls (see [game.end](#), [game.lose](#), [game.win](#)).

Syntax

```
ended game.ended ( );
```

Return Values

ended

Boolean that is only true if the game ended.

Examples

```
// wait for the game to end  
wait( !( game.ended() ) );
```

game.fadeIn

Causes the game to fade in from total black within a given time.

Syntax

```
game.fadeIn ( [time] [, "wait"] );
```

Parameters

`time`

Numerical amount of ticks controlling the fade's duration. If no `time` is passed, the fade happens instantly.

`"wait"`

The script will wait until the fade is done.

Examples

```
// make a 2 seconds fade in  
game.fadeIn( 120, "wait" );
```

game.fadeOut

Causes the game to fade out to total black within a given time.

Syntax

```
game.fadeOut ( [time] [, "wait"] );
```

Parameters

`time`

Numerical amount of ticks controlling the fade's duration. If no `time` is passed, the fade happens instantly.

`"wait"`

The script will wait until the fade is done.

Examples

```
// make a 5 seconds fade out  
game.fadeOut( 300, "wait" );
```

game.getCompletedBefore

Returns if the current level has been completed before.

To test this function in the Modding section of the game use [game.setCompletedBeforeForTesting](#). Does not work in Multiplayer levels.

Syntax

```
completed game.getCompletedBefore ();
```

Return Values

`completed`

Boolean that is only true if the level has been completed before or after a [game.setCompletedBeforeForTesting](#) call within the Modding section of the game.

Examples

```
// check if the level has been beaten before
if( game.getCompletedBefore() )
    // the level has been beaten before
end
```


game.getEvent

Checks if an event that matches the specified name and every filter was posted during the last tick.

Each event will only be returned once unless events are reset by [game.resetEvents](#). Event details can be accessed with [game.getEventValue](#) after a successful [game.getEvent](#) call. A list of available events and their parameters can be found in the [Event Handling](#) section.

All parameters of this function are limited to a maximum length of 61 characters.

Syntax

```
event game.getEvent ( eventName [, filterName1, filterValue1 [...]] );
```

Parameters

`filterName1, filterValue1 [...]`

Pair containing a string for the filter's name and its type unspecified value. A dynamic amount of filter is supported, no filter is required.

Return Values

`event`

Boolean that is only true if an event was found that has the given `eventName` and matches every `filterName/filterValue` pair.

Examples

```
// check if player1 was damaged during a collision
if( game.getEvent( "collision", "objectAName", "player1", "objectADamaged", true ) )
    // player1 was damaged
end
```

```
// check if an enemy was killed during a collision
if( game.getEvent( "collision", "objectAType", "enemy", "objectADestroyed", true ) )
    // an enemy was killed
end
```

```
// check if a player collided with an enemy using a preset called "Team Red Flag":
if( game.getEvent( "collision", "objectAType", "player", "objectBType", "enemy",
"objectBPreset", "Team Red Flag" ) )
    // a player collided with a "Team Red Flag" enemy
end
```

game.getEventValue

Returns the specified event parameter's value after a successful `game.getEvent` call during the current tick.

A list of parameter names for all events can be found in the [Event Handling](#) section.

Syntax

```
value game.getEventValue ( name );
```

Parameters

`name`

String containing the parameter's name.

Return Values

`value`

Parameter's type unspecified value.

Examples

```
// check if a player was killed during a collision
if( game.getEvent( "collision", "objectAType", "player", "objectADestroyed", true ) )
    // a player was destroyed during a collision, check if it was the local player
    if( ==( game.getEventValue ( "objectAName" ), game.getLocalPlayer() ) )
        // local player was destroyed
    else
        // another player was destroyed
    end
end
```

game.getLocalPlayer

Returns the local player's script name.

Syntax

```
name game.getLocalPlayer ();
```

Return Values

name

String containing the local player's script name.

Examples

```
// damage the local player  
asCode( game.getLocalPlayer() ).damage();
```

game.getMinimumScore

Returns the level's minimum score.

Only works for Singleplayer levels with the settings "Game Mode: Score" and "Win Condition: Score Reached".

Syntax

```
score game.getMinimumScore ();
```

Return Values

score

Level's numerical minimum score.

Examples

```
// announce the level's minimum score  
game.message( +( "Minimum Score: ", asScore( game.getMinimumScore() ) ) );
```

game.getPlayer

Returns the specified player's script name.

Syntax

```
name game.getPlayer ( id );
```

Parameters

`id`

Player's numerical id, 0 for the first player.

Return Values

`name`

String containing the player's script name.

Examples

```
// damage the third player  
asCode( game.getPlayer( 2 ) ).damage();
```

game.getPlayerCount

Returns the player count the game has been started with.

Syntax

```
players game.getPlayerCount ();
```

Return Values

players

Numerical amount of players the game has been started with.

Examples

```
// announce the player count in a message  
game.message( +( game.getPlayerCount(), " players are competing!" ) );
```

game.getScoreLimit

Returns the Multiplayer game's score limit defined by the lobby host.

Syntax

```
limit getScoreLimit ();
```

Return Values

`limit`

Multiplayer game's numerical score limit. If no score limit has been set, this is 0.

Examples

```
// announce the score limit  
game.message( +( "Scorelimit: ", asScore( game.getScoreLimit() ) ) );
```

game.getSkip

Returns if a player has pressed the skip button during a skipable section (see [game.setSkipable](#)) during the last tick.

It is advised to call this function in a parallel thread (see [startThread](#)).

Syntax

```
skip getSkip ();
```

Return Values

skip

Boolean that is only true if a player has pressed skip during a skipable section.

Examples

```
// player won, there is an outro in the main thread and skipping is done in this parallel thread

// enter skipable section
game.setSkipable( true );

// wait for the player to press skip
wait( !( game.getSkip() ) );

// end the level
game.win();
```


game.getTeam

Returns the specified teams's script name.

Syntax

```
name game.getTeam ( id );
```

Parameters

`id`

Teams's numerical id, 0 for the first team.

Return Values

`name`

String containing the team's script name.

Examples

```
// add one point to the first team's score bonus  
asCode( game.getTeam( 0 ) ).addScoreBonus( 1 );
```

game.getTeamCount

Returns the current team count.

Singleplayer levels always have 1 team. In Multiplayer the initial team count depends on the lobby settings: If a game with 4 teams is started, 4 teams are created. FFA creates 1 team per player lobby slot. This handling may result in empty teams after game start.

Syntax

```
count getTeamCount ( );
```

Return Values

count

Current numerical team count.

Examples

```
// announce the team count in a message  
game.message( +( game.getTeamCount(), " teams are competing!" ) );
```

game.getTime

Returns the current game time in ticks.

Syntax

```
time game.getTime ();
```

Return Values

time

Current numerical game time in ticks.

Examples

```
// wait until the first minute is over  
wait( <=( game.getTime(), 3600 ) );
```

game.getTimeLimit

Returns the Multiplayer game's time limit defined by the lobby host.

Syntax

```
limit getTimeLimit ();
```

Return Values

limit

Multiplayer game's time limit. If no time limit has been set, this is 0.

Examples

```
// announce the time limit  
game.message( +( "Timelimit: ", asTime( game.getTimeLimit() ) ) );
```

game.isFading

Returns if the game is currently fading to or from black.

Syntax

```
fading game.isFading ();
```

Return Values

`fading`

Boolean that is only true if the game is currently fading.

Examples

```
// wait for the fade to be done  
wait( game.isFading() );
```

game.isFFA

Returns if the game has been started as FFA.

Singleplayer games count as FFA.

Syntax

```
ffa game.isFFA ();
```

Return Values

ffa

Boolean that is only true if the game has been started as FFA.

Examples

```
// check if the game has been started as free for all
if( game.isFFA () )
    // game has been started as FFA
end
```

game.killAllEnemies

Instantly kills all enemies and shots.

Every killed element will play its standard death animation. Enemies killed will not award any points to any player. Actions that are configured to happen when the enemy is "Being Defeated" within the enemy's action management will be triggered.

Syntax

```
game.killAllEnemies ();
```

Examples

```
// kill all enemies  
game.killAllEnemies();
```

game.killEnemiesOfSpawner

Instantly kills all enemies that are assigned to a given spawner.

Every killed enemy will play its standard death animation. Enemies killed will not award any points to any player. Actions that are configured to happen when the enemy is "Being Defeated" within the enemy's action management will be triggered.

Syntax

```
game.killEnemiesOfSpawner ( spawner );
```

Parameters

`spawner`

String containing the spawner's script name.

Examples

```
// kill all enemies assigned to the spawner named "spawner1"  
game.killEnemiesOfSpawner( "spawner1" );
```


game.lose

Causes a Singleplayer game to be lost.

Syntax

```
game.lose ();
```

Examples

```
// player lost  
game.lose();
```

game.message

Displays a given message in the local player's chat.

Syntax

```
game.message ( text );
```

Parameters

text

String containing the message.

Examples

```
// welcome the player  
game.message( "Welcome!" );
```

game.pauseTime

Pauses the game time.

Elements are still able to move and act. The player's multiplier will not decrease.

Syntax

```
game.pauseTime ();
```

Examples

```
// pause time and wait for the player to reach the player detector named "start"
game.pauseTime();
wait( !( start.get() ) );

// the player reached start, resume the game time
game.resumeTime();
```

game.ping

Pings a given world position.

The ping is the same as one caused by a ping element.

Syntax

```
game.ping ( x, y );
```

Parameters

x, *y*

Ping's numerical world coordinates.

Examples

```
// ping the player's current position  
game.ping( player1.getPos() );
```

game.playMusic

Starts a given track with a given transition or stops music.

If the given track is already playing, nothing will happen.

Syntax

```
game.playMusic ( track, [transMode, transDuration] );
```

Parameters

track

Either a string containing a track name or a numerical value referencing to those names:

- **0:** Stops music
- **1:** "Expanding Universe"
- **2:** "Space Elevator"
- **3:** "Inhale, Exhale"
- **4:** "Dystopia"
- **5:** "Message from HQ"
- **6:** "The Swarm"
- **7:** "Spheres"
- **8:** "Infestation"
- **9:** "Artificial Intelligence"
- **10:** "Cryosphere"

transMode

String controlling the transition mode. Must be one of the following:

- **"hard"**: Instantly ends the current track and starts the given [track](#) without fading.
- **"fadeOutHardIn"**: Fades out the current track and starts the given [track](#) without fading.
- **"crossfade"**: Performs a crossfade of the current track and the given [track](#).
- **"fadeOutIn"**: Fades out the current track and starts a fade in with the given [track](#).

transDuration

Numerical amount of ticks controlling the transition's duration. The maximum duration is 6000 ticks.

Examples

```
// switch to "Artificial Intelligence" and wait 10 seconds
game.playMusic( "Artificial Intelligence", "crossfade", 180 );
wait( 600 );

// restart the same track (needs to be stopped first)
game.playMusic( 0, "hard" );
game.playMusic( 9, "hard" );
```

game.resetEvents

Resets all current events to be able to receive them via [game.getEvent](#) again.

Syntax

```
game.resetEvents ();
```

Examples

```
// reset all events  
game.resetEvents();
```

game.resumeTime

Resumes the game time if it is paused.

Syntax

```
game.resumeTime ();
```

Examples

```
// pause game time and start a countdown
game.pauseTime();
bot.say( "Ready!", "wait" );
bot.say( "Set!", "wait" );

// start
bot.say( "Go!" );
game.resumeTime();
```

game.scheduleCheckpointLoad

Causes the game to load the last checkpoint as soon as all scripts are handled for the current tick.

If no checkpoint has been reached yet, the level will restart. Does not work in Multiplayer levels.

Syntax

```
game.scheduleCheckpointLoad ();
```

Examples

```
// load the last checkpoint or restart if there wasn't any  
game.scheduleCheckpointLoad();
```


game.scheduleCheckpointSave

Causes the game to save a new checkpoint as soon as all scripts are handled for the current tick.

Does not work in Multiplayer levels.

Syntax

```
game.scheduleCheckpointSave ( [x, y] );
```

Parameters

[x, y]

Specifies the player's numerical spawn position for this checkpoint. If no [x, y] is given, the current player's swarm position will be used. The checkpoint's position has to be inside of at least one field wall and outside of every obstacle wall.

Examples

```
// save a checkpoint at a script position named "checkpoint"  
game.scheduleCheckpointSave( checkpoint.getPos() );
```

game.setBars

Shows or hides the black cinematic bars.

Syntax

```
game.setBars ( state [, instantly] );
```

Parameters

state

Boolean to show (true) or hide (false) the black cinematic bars.

instantly

If true, the bar's slide animation will be skipped.

Examples

```
// slide in the bars and wait 3 seconds
game.setBars( true );
wait( 180 );

// hide them instantly
game.setBars( false, true );
```

game.setCompletedBeforeForTesting

Forces [game.getCompletedBefore](#) to return the given value.

Does only work in the Modding section of the game. Does not work in Multiplayer levels.

Syntax

```
game.setCompletedBeforeForTesting ( completed );
```

Parameters

completed

Boolean that will be returned by [game.getCompletedBefore](#).

Examples

```
// cause the script in the modding section to act like the level has been completed before  
game.setCompletedBeforeForTesting( true );
```

game.setEvECollisionEvents

Tells the game whether it has to post "enemy vs. enemy" collision events or not.

Per default those events are not posted as it might slow down performance in levels with many enemies.

Syntax

```
game.setEvECollisionEvents ( post );
```

Parameters

post

Boolean to post (true) or block (false) "enemy vs. enemy" collision events.

Examples

```
// activate "enemy vs. enemy" collision events  
game.setEvECollisionEvents( true );
```

game.setSkipable

Tells the game that a skipable section has been entered or left.

While in a skipable section, the game shows "Skip: X" in the lower right corner of the screen automatically (X being configured skip button). If the player presses the skip button within a skipable section, a white flash will automatically be shown. The script has 1 tick to change the whole scene as desired without the player seeing it. The script itself has to constantly check [game.getSkip](#) and to react accordingly.

Syntax

```
game.setSkipable ( skipable );
```

Parameters

skipable

Boolean to enter (true) or leave (false) a skipable section.

Examples

```
// enter skipable section
game.setSkipable( true );

// wait for the user to skip
wait( ! ( game.getSkip() ) );

// move the player to the script position called "finish" and update the camera instantly
player1.setPos( finish.getPos() );
cam.setPos( player1.getPos() );

// leave skipable section
game.setSkipable( false );
```

game.setTeamCount

Changes the team count.

If the current team count is lower than the given one, empty teams will be created. If the current team count is higher than the given one, this will delete as many teams as required. The function always starts to delete teams from the back. Only empty teams can be deleted. If any of the teams that would have to be deleted has at least one player assigned to it, the function will delete no team at all. Does not work in Singleplayer.

Syntax

```
game.setTeamCount ( count );
```

Parameters

`count`

Numerical value for the new team count ranging from 1 to 8.

Examples

```
// set the team count to 3  
game.setTeamCount( 3 );
```

game.setTime

Sets the current game time to a given value.

Does not work if the game has already ended (see [game.ended](#)).

Syntax

```
game.setTime ( time );
```

Parameters

time

Numerical value in ticks for the new game time ranging from 0 to 359940 (99 minutes and 99 seconds).

Examples

```
// reset game time  
game.setTime( 0 );
```

game.showScore

Shows or hides the score for the local player.

If the level is using "PvE score" (see [game.usePvEScore](#)) the local player's multiplier is also shown or hidden accordingly.

Syntax

```
game.showScore ( show );
```

Parameters

`show`

Boolean to show (true) or hide (false) the local player's score.

Examples

```
// show score  
game.showScore( true );
```


game.showTime

Shows or hides the time for the local player.

Syntax

```
game.showTime ( show );
```

Parameters

show

Boolean to show (true) or hide (false) the local player's time.

Examples

```
// hide time  
game.showTime( false );
```

game.usePvEScore

Activates or deactivates the use of "Player Versus Environment" score.

If it is active, killing enemies (excluding opponent's swarms) rewards the player with points which are also multiplied by the current player's multiplier. Additionally the multiplier is raised by collecting enemies' drops. By default this score type is only activated in Singleplayer levels using game type "Score".

Syntax

```
game.usePvEScore ( use );
```

Parameters

`use`

Boolean to activate (true) or deactivate (false) the use of "Player Versus Environment" score.

Examples

```
// activate PvE score  
game.usePvEScore( true );
```

game.usePvPScore

Activates or deactivates the use of "Player Versus Player" score.

If it is active, killing an opponent's swarm rewards the player with a single point. By default this score type is activated only in Multiplayer levels. Does not work in Singleplayer levels.

Syntax

```
game.usePvPScore ( use );
```

Parameters

`use`

Boolean to activate (true) or deactivate (false) the use of "Player Versus Player" score.

Examples

```
// deactivate PvP score  
game.usePvPScore( true );
```

game.win

Causes a Singleplayer game to be won.

Syntax

```
game.won ();
```

Examples

```
// player won  
game.won();
```

Global

and / **&&**

Performs a logical AND with all given parameters.

Syntax

```
result and ( value1, value2 [...] );
```

```
result && ( value1, value2 [...] );
```

Parameters

`value1, value2 [...]`

Dynamic list of Booleans for the logical AND operation.

Return Values

`result`

true if every `value` is true, otherwise false.

Examples

```
// check if both variables named "a" and "b" are true
if( &&( a.get(), b.get() ) )
    // both variables are true
end
```

asCode

Interprets the given string's value as code and executes it.

Syntax

```
execute asCode ( code );
```

Parameters

`code`

String containing the code to be executed.

Return Values

`execute`

`code`'s return value after being executed by this function.

Examples

```
// add a white marker to the first player  
asCode( game.getPlayer( 0 ) ).addMarker( 1, 1, 1 );
```

asScore

Returns a formatted score string for a given amount of points.

Syntax

```
score asScore ( points );
```

Parameters

points

Numerical amount of points.

Return Values

score

Formatted score string like "100.000".

Examples

```
// announce the player's score  
game.message( asScore( player1.getScore() ) );
```


asString

Returns a given parameter interpreted as string.

Can be used to translate Booleans or numbers.

Syntax

```
string asString ( value );
```

Parameters

`value`

Boolean or number to be interpreted as string.

Return Values

`string`

`value` interpreted as string.

Examples

```
// announce the player's current swarm level  
game.message( asString( player1.getLevel() ) );
```

asTime

Returns a formatted time string for a given amount of ticks.

Syntax

```
time asTime ( ticks );
```

Parameters

`ticks`

Numerical amount of ticks.

Return Values

`time`

Formatted score string like "02:30.000".

Examples

```
// announce the current time played  
game.message( asTime( game.getTime() ) );
```

create

Creates a specified script object.

Syntax

```
create ( "Bot", name [, x, y, z [, primR, primG, primB [, secR, secG, secB]]] );  
create ( "Enemy", name, preset, x, y [, silent [, angle]] );  
create ( "Var", name [, value1 [...]] [, scope] );
```

Parameters

name

String containing the script object's name. When creating a bot, its display name will also be set to this value (see [bot.setName](#)). If an empty name is passed, a unique name will be selected. Interaction with the script object is not possible this way.

x, y, z

Numerical coordinates to set the object's world position. When creating an enemy, the coordinates need to be within at least one field wall and outside of all obstacle walls. The coordinates 0, 0, 0 are used as a default for a bot.

primR, primG, primB

Numerical RGB color for the bot's eye and name. Each value ranges from 0.0 to 1.0. Bot-7's primary color 0.4235, 1, 0.7529 is used as a default.

secR, secG, secB

Numerical RGB color for the bot's body, hair and message box. Each value ranges from 0.0 to 1.0. Bot-7's secondary color 0.4235, 1, 0.7529 is used as a default.

preset

Case sensitive string containing the enemy's preset's name.

silent

Boolean that controls whether the enemy should be spawned silently and instantly (true) or play its standard spawn animation (false). Enemies spawn with their standard spawn animation per default.

angle

Enemy's look direction in degrees ranging from 0 to 360. 0 means right, the angle goes counter clockwise. Enemies look up per default (**angle** = 90). May be overruled by the enemy's own targeting

value1 [...]

Variable's initial value. Can also be a list of parameters to create an array. Variables are empty per default.

scope

Must be one of the following:

- **"global"**: The variable can be accessed from every script. This is the default scope.
- **"local"**: The variable can only be accessed by the script that created it.
- **"persistent"**: The variable keeps its value when a checkpoint is loaded. Persistent variables are always also global.

Examples

```
// create a standard bot  
create( "Bot", "bot", 0, 300, -100 );
```

```
// create an enemy using the "Meteor" preset directly on the player's position  
create( "Enemy", "e", "Meteor", player1.getPos() );
```

```
// create a persistent variable to count all player deaths  
create( "Var", "deathCount", 0, "persistent" );
```

```
// create an array to store the lives of three players  
create( "Var", "playerLivesLeft", 5, 5, 5 );
```

delete

Deletes a given script object.

The script's local scope is searched for an object with the given name first. If no object is found, the global scope is searched. If the object is an enemy it will neither play any death animation nor will any points be awarded to any player.

Syntax

```
delete ( object );
```

Parameters

object

String containing the script object's name.

Examples

```
// remove a variable called "temp"  
delete( "temp" );
```

endScript

Ends every script with a given file name.

May end the script that called this function.

Syntax

```
endScript ( script );
```

Parameters

script

String containing the script's file name including the ".txt" extension.

Examples

```
// end all "eventHandling.txt" scripts  
endScript( "eventHandling.txt" );
```

exists

Checks if at least one of a dynamic number of given script objects exists.

Syntax

```
exist exists ( object1 [...] );
```

Parameters

`object1 [...]`

String containing the script object's name. At least one name is needed but the function supports a dynamic amount.

Return Values

`exist`

Boolean that is only false if none of the given `objects` exist.

Examples

```
// check if at least one of three enemies named "enemy1", "enemy2" and "enemy3" exist
if( exists( "enemy1", "enemy2", "enemy3" ) )
    // at least one of those enemies is still alive
end
```

exit

Terminates the script that called this function.

Syntax

```
exit ();
```

Examples

```
// terminate the script  
exit();
```


if

Controls code branching based on a given condition.

Ifs can be nested with other ifs or whiles. No semicolon is required.

Syntax

```
if ( condition )  
  [trueBlock]  
end
```

```
if ( condition )  
  [trueBlock]  
else  
  [falseBlock]  
end
```

Parameters

`condition`

Boolean that controls what block should be executed.

`trueBlock`

Block of code that gets executed if `condition` is true.

`falseBlock`

Block of code that gets executed if `condition` is false.

Examples

```
// create a variable named "a" and store the number 5  
create( "Var", "a", 5 );  
  
// check if "a" equals 5 (it does)  
if( ==( a.get(), 5 ) )  
  // "a" equals 5, this code WILL be executed  
end  
  
// check if "a" equals 3 (it does not as a equals 5)  
if( ==( a.get(), 3 ) )  
  // this code will NOT be executed  
end
```

```
// create a variable named "a" and store the number 5
create( "Var", "a", 5 );

// check if "a" is greater than 1 (it is as 5 > 1)
if( >( a.get(), 1 ) )
    // 5 is greater than 1, this code WILL be executed
else
    // this code will NOT be executed
end

// check if "a" is smaller than 2 (it is not as 5 > 2)
if( <( a.get(), 2 ) )
    // this code will NOT be executed
else
    // 5 is not smaller than 2, this code WILL be executed
end
```

import

Imports the code of a given script right after the import call.

Unlike [startThread](#), this does not start a new script. The code will be part of the same namespace that belongs to the script that called this function. This call will not be logged.

Syntax

```
import ( script );
```

Parameters

script

String containing the script's file name including its ".txt" extension.

Examples

```
// import the script code stored in the file "playerHit.txt"
import( "playerHit.txt" );
```

not** / **!

Performs a logical NOT with a given Boolean

Syntax

```
return not ( value );
```

```
return ! ( value );
```

Parameters

`value`

Boolean for the logical NOT operation.

Return Values

`return`

Boolean that is true unless `value` is true.

Examples

```
// check if no script object named "enemy1" exists
if( !( exists( "enemy1" ) ) )
    // no script object named "enemy1" exists
end // if
```

or / *||*

Performs a logical OR with all given parameters.

Syntax

```
result or ( value1, value2 [...] );
```

```
result || ( value1, value2 [...] );
```

Parameters

`value1`, `value2` [...]

Dynamic list of Booleans for the logical OR operation.

Return Values

`result`

true if at least one `value` is true, otherwise false.

Examples

```
// check if at least one of the two variables named "a" or "b" is true
if( ||( a.get(), b.get() ) )
    // at least one of the two variables is true
end
```

setLogging

Activates or deactivates the current script's logging.

The log can be found in your documents folder under "\\Dedication Games\Swarm Universe\[YOUR STEAM ID]\log.txt". Logging does only work in the Modding section of the game.

Syntax

```
setLogging ( log );
```

Parameters

`log`

Boolean to activate (true) or deactivate (false) the current script's logging.

Examples

```
// activate logging for this script  
setLogging( true );
```

startThread

Starts a new script as parallel thread using its own namespace.

Syntax

```
startThread ( script );
```

Parameters

script

String containing the script's file name including its ".txt" extension.

Examples

```
// start a parallel thread with the script saved in "eventHandling.txt"  
startThread( "eventHandling.txt" );
```

wait

Causes the script to wait.

A script can only process 1000 script lines between two wait calls. After 1000 script lines without waiting the script will wait automatically for one tick.

Syntax

```
wait ( [time] );
```

Parameters

`time`

Either a numerical amount of ticks or a function call that returns a Boolean. If a call is passed, the script will wait until the call returns false. If no `time` is passed, the script will wait for one tick.

Examples

```
// wait for the player to dash
wait( !( player1.isDashing() ) );

// wait 2 more seconds
wait( 120 );
```


while

Loops and executes the code between the while and its according end again and again as long as the given condition is true.

Whiles can be nested with other whiles or ifs. No semicolon is required.

Syntax

```
while ( condition )  
  [block]  
end
```

Parameters

`condition`

Boolean that controls if the `block` of code should be executed.

`block`

Block of code that gets executed if `condition` is true.

Examples

```
// count down from 3 to 1  
create( "Var", "count", 3 );  
  
while( >( count.get(), 0 ) )  
  // announce current count  
  game.message( count.asString() );  
  
  // lower count  
  count.+=( 1 );  
  
  // wait one second  
  wait( 60 );  
end
```

XOR

Performs a logical XOR with all given parameters.

Syntax

```
result xor ( value1, value2 [...] );
```

Parameters

`value1, value2 [...]`

Dynamic list of Booleans for the logical XOR operation.

Return Values

`result`

true if exactly one `value` is true, otherwise false.

Examples

```
// check if exactly one of the two variables named "a" and "b" is true
if( xor ( a.get(), b.get() ) )
    // exactly one of the two variables is true
end
```

+

Adds up a dynamic amount of numbers or strings.

Syntax

```
sum + ( value1, value2 [...] );
```

Parameters

`value1, value2 [...]`

Values that will be added up. At least 2 values are needed but the function supports a dynamic amount.

Return Values

`sum`

If all `values` are numbers, this is the numerical result of their addition. Otherwise this is a string containing every `value` interpreted as string and put together.

Examples

```
// add up the score of two rounds saved in variables called "scoreR1" and " scoreR2"  
score.set ( +( scoreR1.get(), scoreR2.get() ) );
```

-

Returns the subtraction's result of two given numbers.

Syntax

```
result - ( value1, value2 );
```

Parameters

`value1`

Numerical value `value2` will be subtracted off.

`value2`

Numerical value that will be subtracted from `value1`.

Return Values

`result`

Subtraction's numerical result of `value1` minus `value2`.

Examples

```
// calculate a player's score based on variables named "kills" and "deaths"  
player1.setScore( -( kills.get(), deaths.get() ) );
```

*

Multiplies a dynamic amount of numbers.

Syntax

```
product * ( factor1, factor2 [...] );
```

Parameters

`factor1`, `factor2 [...]`

Numerical values that will be multiplied. At least 2 values are needed but the function supports a dynamic amount.

Return Values

`product`

Multiplication's numerical result.

Examples

```
// calculate a player's score based on variables named "points" and "multipliers"  
player1.setScore( *( points.get(), multiplier.get() ) );
```



Returns the division's result of two given numbers.

Syntax

```
quotient / ( dividend, divisor );
```

Parameters

`dividend`

Division's numerical dividend that will be divided by the `divisor`.

`divisor`

Division's numerical divisor the `dividend` will be divided by.

Return Values

`quotient`

Division's numerical result of the `divident` divided by the `divisor`.

Examples

```
// calculate the player1's average score per second and save it in a variable named "sps"  
sps.set( *( /( player1.getScore(), game.getTime() ), 60 ) );
```



Checks if the given first number is greater than the given second one.

Syntax

```
result > ( value1, value2 );
```

Parameters

`value1`

First numerical value.

`value2`

Second numerical value.

Return Values

`result`

Boolean that is true if `value1` is greater than `value2`, otherwise false.

Examples

```
// check if team1 has won against team2
if( >( team1.getScore(), team2.getScore() ) )
    // team1 has won
end
```



Checks if the given first number is lower than the given second one.

Syntax

```
result < ( value1, value2 );
```

Parameters

`value1`

First numerical value.

`value2`

Second numerical value.

Return Values

`result`

Boolean that is true if `value1` is lower than `value2`, otherwise false.

Examples

```
// check if there is a new round record based on the variables named "round" and "best"
if( <( round.get(), best.get() ) )
    // there is a new round record
end
```


>=

Checks if the given first number is greater than or equal to the given second one.

Syntax

```
result >= ( value1, value2 );
```

Parameters

`value1`

First numerical value.

`value2`

Second numerical value.

Return Values

`result`

Boolean that is true if `value1` is greater than or equal to `value2`, otherwise false.

Examples

```
// check if the player's score is high enough based on a variable named "minimumScore"
if( >=( player1.getScore(), minimumScore.get() ) )
    // the player's score is high enough
end
```



Checks if the given first number is lower than or equal to the given second one.

Syntax

```
result >= ( value1, value2 );
```

Parameters

`value1`

First numerical value.

`value2`

Second numerical value.

Return Values

`result`

Boolean that is true if `value1` is lower than or equal to `value2`, otherwise false.

Examples

```
// check if the target time has been beaten based on a variable named "targetTime"
if( <=( game.getTime(), targetTime.get() ) )
    // the target time has been beaten
end
```



Checks if all given values are equal to each other.

Syntax

```
equal == ( value1, value2 [...] );
```

Parameters

`value1, value2 [...]`

Values to be compared with each other. At least 2 values are needed but the function supports a dynamic amount.

Return Values

`equal`

Boolean that is only true if all `values` are equal to each other.

Examples

```
// check if the 2 player game is a draw
if( ==( player1.getScore(), player2.getScore() ) )
    // the game is a draw
end
```



Checks if at least one given value is different to the others.

Syntax

```
unequal == ( value1, value2 [...] );
```

Parameters

`value1, value2 [...]`

Values to be compared with each other. At least 2 values are needed but the function supports a dynamic amount.

Return Values

`unequal`

Boolean that is only true if at least one `value` is different to the others.

Examples

```
// check if the player count differs from 3
if( !=( game.getPlayerCount(), 3 ) )
    // the player count is not 3
end
```

Logic

This section's function set applies to the following elements: Script Output, Player Detector, Spawn Tracker, Timer, And, Or, Not, Switch, Delay, Extender and Change Detector.

logic.get

Returns the logic element's last output.

Syntax

```
output logic.get ();
```

Return Values

output

Boolean for the logic element's last output.

Examples

```
// check if a timer named "timer1" was active during the last tick
if( timer1.get() )
    // "timer1" was active
end
```

logic.set

Changes the logic element's output.

Syntax

```
logic.set ( style );
```

Parameters

style

Either a Boolean to set the logic element's output or a string containing "normal" to reactivate the logic element's standard return behavior. If a Boolean is passed, the logic element's background handling will still be performed. For example: A timer ticks down even if it is forced to return true.

Examples

```
// deactivate a player detector named "detector1" for 2 seconds
detector1.set( false );
wait( 120 );

// reactivate the player detector's normal behavior
detector1.set( "normal" );
```

logic.trigger

Causes the logic element to return true during the upcoming tick.

Syntax

```
logic.trigger ();
```

Examples

```
// force a switch named "button" to return true during the upcoming tick  
button.trigger();
```


Math

math.abs

Returns the absolute value of a given number.

Syntax

```
absolute math.abs ( value );
```

Parameters

value

Numeric value.

Return Values

absolute

Absolute *value*.

Examples

```
// calculate the difference between two scores  
difference.set( math.abs( -( player1.getScore(), player2.getScore() ) ) );
```

math.atan

Returns a given value's arc tangent in radians.

Arc tangent is the inverse operation of tangent. The tangent's value does not determine with certainty in which quadrant the angle falls (see [math.atan2](#)).

Syntax

```
arc math.atan ( value );
```

Parameters

value

Numerical value.

Return Values

arc

Numerical arc tangent of *value* in radians.

Examples

```
// calculate and save the arc tangent of 0.5 in a variable named "arc"  
arc.set( math.atan( 0.5 ) );
```

math.atan2

Returns the two given coordinates' arc tangent in radians.

The coordinates' signs are taken into account to determine the correct quadrant.

Syntax

```
arc math.atan2 ( x, y );
```

Parameters

x, y

Numerical coordinates.

Return Values

arc

Numerical arc tangent of *x*, *y* in radians.

Examples

```
// calculate and save the arc tangent of 20/20 in a variable named "arc"  
arc.set( math.atan2( 20, 20 ) );
```

math.ceil

Rounds up a given number to its next integral value.

Syntax

```
round math.ceil ( value );
```

Parameters

`value`

Numerical value to be rounded up.

Return Values

`round`

Numerical result of `value` being rounded up.

Examples

```
// round up the variable named "points"  
points.set( math.ceil( points.get() ) );
```

math.cos

Returns the cosine of a given angle.

Syntax

```
cosine math.cos ( angle );
```

Parameters

angle

Numerical angle expressed in radians.

Return Values

cosine

Numerical cosine of *angle*.

Examples

```
// calculate and save the cosine of pi in a variable named "cosine"  
cosine.set( math.cos( math.pi() ) );
```

math.floor

Rounds down a given number to its previous integral value.

Syntax

```
round math.floor ( value );
```

Parameters

`value`

Numerical value to be rounded down.

Return Values

`round`

Numerical result of `value` being rounded down.

Examples

```
// round down the variable named "points"  
points.set( math.floor( points.get() ) );
```

math.length

Returns a vector's length.

Syntax

```
len math.length ( x, y [, z] );
```

Parameters

x, y, z

Numerical coordinates of a 2d or 3d vector.

Return Values

len

Numerical vector's length.

Examples

```
// calculate the distance between two players and save it in a variable named "distance"
create( "Var", "pos1", player1.getPos() );
create( "Var", "pos2", player2.getPos() );
distance.set( math.length( -( pos1.get( 0 ), pos2.get( 0 ) ), -( pos1.get( 1 ), pos2.get( 1 ) ) ) );
```


math.mod

Returns the division's remainder of two given numbers.

Syntax

```
remainder math.mod ( dividend, divisor );
```

Parameters

dividend

Division's numerical dividend that will be divided by the *divisor*.

divisor

Division's numerical divisor the *dividend* will be divided by.

Return Values

quotient

Numerical remainder of *divident* divided by *divisor*.

Examples

```
// check if a division with variables named "a" and "b" has no remainder
if( ==( math.mod( a.get(), b.get() ), 0 ) )
    // "a" divided by "b" has no remainder
end
```

math.pi

Returns pi.

Syntax

```
π math.pi ( );
```

Return Values

π

Numerical value of pi.

Examples

```
// announce pi  
game.message( +( "Pi is ", math.pi() ) );
```

math.pow

Returns the result of a mathematical expansion (x^y).

Syntax

```
expansion math.pow ( base, exponent );
```

Parameters

base

Expansion's numerical base that will be raised to the power of *exponent*.

exponent

Expansion's numerical exponent's power *base* will be raised to.

Return Values

expansion

Numerical result of *base* raised to the power of *exponent*.

Examples

```
// use Pythagoras' theorem to calculate the hypotenuse of a right triangle (based on variables  
names "a" and "b") and store the result in a variable named "c"  
c.set( math.sqrt( +( math.pow( a.get(), 2 ), math.pow( b.get(), 2 ) ) ) );
```

math.rnd

Returns a random number within a given range.

Syntax

```
random math.rnd ( max );
```

```
random math.rnd ( min, max );
```

Parameters

`max`

Exclusive numerical maximum.

Return Values

`min`

Inclusive numerical minimum. If no `min` is passed, the minimum will be 0.

Examples

```
// get and save a random number between 0 and 10 (both inclusive) in a variable named "num"  
num.set( math.rnd( 11 ) );
```

```
// get and save a random number between 5 and 10 (both inclusive) in a variable named "num"  
num.set( math.rnd( 5, 11 ) );
```

math.sin

Returns the sine of a given angle.

Syntax

```
sine math.sin ( angle );
```

Parameters

angle

Numerical angle expressed in radians.

Return Values

sine

Numerical sine of *angle*.

Examples

```
// calculate and save the sine of 1.5 pi in a variable named "sine"  
sine.set( math.sin( *( math.pi(), 1.5 ) ) );
```

math.sqrt

Returns the given number's square root.

Syntax

```
root math.sqrt ( num );
```

Parameters

`num`

Numerical value.

Return Values

`root`

Numerical square root of `num`.

Examples

```
// calculate and save the square root of a variable named "number" in a variable named "root"  
root.set( math.sqrt( number.get() ) );
```

math.tan

Returns the tangent of a given angle.

Syntax

```
tangent math.tan ( angle );
```

Parameters

angle

Numerical angle expressed in radians.

Return Values

tangent

Numerical tangent of *angle*.

Examples

```
// calculate and save the tangent of pi/4 in a variable named "tangent"  
tangent.set( math.tan( /( math.pi(), 4 ) ) );
```

math.toDeg

Converts a given angle's radians to degrees.

Syntax

```
degrees math.toDeg ( radians );
```

Parameters

radians

Numerical angle expressed in radians.

Return Values

degrees

Numerical conversion from *radians* to degrees.

Examples

```
// convert pi radians to degrees (180)
math.toDeg( math.pi() );
```


math.toRad

Converts a given angle's degrees to radians.

Syntax

```
radians math.toRad ( degrees );
```

Parameters

degrees

Numerical angle expressed in degrees.

Return Values

radians

Numerical conversion from *degrees* to radians.

Examples

```
// convert 90 degrees to radians (1/2 pi)  
math.toRad( 90 );
```

math.trunc

Truncates everything behind the coma of a given value and returns the outcome.

Syntax

```
truncated math.trunc ( value );
```

Parameters

value

Numerical value.

Return Values

round

Numerical outcome after truncating everything behind *value*'s comma.

Examples

```
// truncate everything behind the comma in a variable named "a"  
a.set( math.trunc( a.get() ) );
```

Player

player.addMarker

Adds a given RGB color marker to the player's swarm.

If the player's swarm is not alive, it already has the given marker or it already has the maximum of 3 markers, nothing will happen. Markers for bonuses ("Speed Boost" and "Dash Boost") can be used but will not grant the respective bonus. A player's swarm cannot pick up a bonus if it already has its marker.

Syntax

```
player.addMarker ( r, g, b );
```

Parameters

r, g, b

Marker's numerical RGB values, each ranging from 0.0 to 1.0.

RGB values for available bonuses are:

- **Speed Boost:** 0.2, 0.25, 1.0
- **Dash Boost:** 1.0, 0.75, 0.33

Examples

```
// add the red keycard marker to player1  
create( "Var", "redKeycard", 1, 0, 0 );  
player1.addMarker( redKeycard.get() );
```

player.addScore

Adds the given amount of points to the player's current score.

If the resulting score is > 999.999.999, the score gets set to 999.999.999. If the resulting score is < 0, the score gets set to 0. Can be used to subtract a value from the score by passing a negative number. Does not work if the game has ended already (see [game.ended](#)).

Syntax

```
player.addScore ( score );
```

Parameters

score

Numerical amount of points to be added to the player's score.

Examples

```
// add 100 points to the player's current score  
player1.addScore( 100 );
```

player.allowDroneInactivity

Allows or disallows the player's drone inactivity handling.

If allowed, the handling causes all drones to become inactive if the player is not moving for a short period of time. By default, this feature is allowed.

Syntax

```
player.allowDroneInactivity ( allow );
```

Parameters

allow

Boolean to allow (true) or disallow (false) the player's drone inactivity handling.

Examples

```
// disallow the player's drone inactivity handling (drones will not become inactive)  
player1.allowDroneInactivity( false );
```

player.dash

Causes the player's swarm to dash.

The player's swarm needs to be alive and moving, otherwise no dash will be started. If the player is already dashing, nothing will happen.

Syntax

```
player.dash ();
```

Examples

```
// cause the player's swarm to dash  
player1.dash();
```

player.damage

Causes the player's swarm to take a given amount of damage

If the swarm is currently shielded, nothing will happen. Kills the swarm if the damage is higher than its remaining levels or if its emergency shield is not charged. Otherwise the swarm loses the given amount of levels and activates its emergency shield.

Syntax

```
player.damage ( amount );
```

Parameters

`amount`

Numerical amount of damage the player's swarm will take.

Examples

```
// cause the player's swarm to take 2 damage  
player1.damage( 2 );
```


player.getAverageDronePos

Returns the player's swarm's average drone position.

Syntax

```
x, y player.getAverageDronePos ( [shiftX, shiftY] );
```

Parameters

shiftX, shiftY

Numerical coordinates that will be added to x, y to get a position relative to the drone's average position, for example a position to its right.

Return Values

x, y

Current numerical average drone position.

Examples

```
// save the player's current average drone position to a variable named "pos"  
pos.set( player1.getAverageDronePos() );
```

player.getInput

Enables the script to directly access the player's input.

Syntax

```
result player.getInput ( action, mode );
```

Parameters

action

Must be one of the following strings: "shoot", "dash", "up", "right", "down" or "left"

mode

Must be:

- **"digital"**: **result** is true, if the player is currently holding the button/stick/key for the given **action**.
- **"analog"**: **result** states how much an analog button or stick for the given action is pressed and ranges from 0.0 (not pressed) to 1.0 (fully pressed). May be used for every **action** but will only return values between 0.0 and 1.0 for "up", "right", "down" or "left". Inputs for "shoot" and "dash" will always return either 0 or 1.
- **"press"**: **result** is true as soon as the player starts to press the button/stick/key for the given **action**. Therefore **result** is true only once until the player releases the button/stick/key and presses it again.
- **"release"**: **result** is true as soon as the player releases the button/stick/key for the given **action**. Therefore **result** is true only once until the player presses the button/stick/key and releases it again.

Return Values

result

The return value type differs based on the selected **mode**. For "digital", "press" and "release" a Boolean will be returned. For "analog" a number between 0.0 to 1.0.

Examples

```
// wait as long as the player is not pressing the "Shoot" button
wait( !( player1.getInput( "shoot", "press" ) ) );
```

player.getLevel

Returns the player's current swarm level.

Syntax

```
level player.getLevel ();
```

Return Values

level

Player's current numerical swarm level.

Examples

```
// check if both players have the same swarm level
if( ==( player1.getLevel(), player2.getLevel() ) )
    // both players have the same swarm level
end
```

player.getMultiplier

Returns the player's current score multiplier.

Syntax

```
multiplier player.getMultiplier ();
```

Return Values

```
multiplier
```

Player's current numerical score multiplier.

Examples

```
// check if the player's multiplier is a new record
if( >( player1.getMultiplier(), bestMultiplier.get() ) )
    // the multiplier is a new record
end
```

player.getPos

Returns the player's swarm's current world position.

Syntax

```
x, y player.getPos ( [shiftX, shiftY] );
```

Parameters

shiftX, shiftY

Numerical coordinates that will be added to x, y to get a position relative to the swarm, for example a position to its right.

Return Values

x, y

Swarm's current numerical world position.

Examples

```
// move the swarm down  
player1.moveTo( player1.getPos( 0, -100 ), "wait" );
```

player.getScore

Returns the player's current score.

Syntax

```
score player.getScore ();
```

Return Values

score

Player's current numerical score.

Examples

```
// check if player1 is leading against player2
if( >( player1.getScore(), player2.getScore() ) )
    // player1 is leading
end
```

player.getSteamName

Returns the player's Steam name.

Syntax

```
name player.getSteamName ();
```

Return Values

name

String containing the player's Steam name.

Examples

```
// welcome the local player  
game.message( +( "Welcome, ", asCode( game.getLocalPlayer() ).getSteamName(), "!" ) );
```

player.getTeam

Returns the player's team's script name.

Syntax

```
name player.getTeam ( );
```

Return Values

name

String containing the team's script name.

Examples

```
// announce the current second team's player count  
game.message( +( asCode( game.getTeam( 1 ) ).getPlayerCount(), " players in team 2." ) );
```


player.hasMarker

Returns if the player has the given RGB color marker.

Syntax

```
has player.hasMarker ( r, g, b );
```

Parameters

r, g, b

Marker's numerical RGB values, each ranging from 0.0 to 1.0.

Return Values

has

Boolean that is only true if the player has the given *r*, *g*, *b* marker.

Examples

```
// check if the player does have the red marker
if( player1.hasMarker( 1, 0, 0 ) )
    // the player has the red marker
end
```

player.isAlive

Returns if the player's swarm is currently alive.

Syntax

```
alive player.isAlive ();
```

Return Values

alive

Boolean that is only true if the player's swarm is currently alive.

Examples

```
// check if the player's swarm is currently alive
if( player1.isAlive() )
    // the swarm is alive
end
```

player.isDashing

Returns if the player's swarm is currently dashing.

Syntax

```
dashing player.isDashing ();
```

Return Values

dashing

Boolean that is only true if the player's swarm is currently dashing.

Examples

```
// wait for the player to stop dashing  
wait( player1.isDashing() );
```

player.isMoving

Returns if the player's swarm is currently moving.

Syntax

```
moving player.isMoving ();
```

Return Values

`moving`

Boolean that is only true if the player's swarm is currently moving.

Examples

```
// wait for the player to stop moving  
wait( player1.isMoving() );
```

player.isShielded

Returns if the player's swarm is currently shielded by its spawn protection or emergency shield.

Does not consider the player's script invincibility (see [player.setInvincibility](#)).

Syntax

```
shielded player.isShielded ();
```

Return Values

shielded

Boolean that is only true if the player's swarm is currently shielded.

Examples

```
// wait for the player to stop being shielded  
wait( player1.isShielded() );
```

player.isShooting

Returns if the player's swarm is currently shooting.

Syntax

```
shooting player.isShooting ();
```

Return Values

shooting

Boolean that is only true if the player's swarm is currently shooting.

Examples

```
// wait for the player to start shooting  
wait( !( player1.isShooting() ) );
```

player.isSwinging

Returns if the player's swarm is currently swinging.

A swarm is swinging if it is moving and all its drones are close together. A swarm without drones can never be swinging.

Syntax

```
swinging player.isSwinging ();
```

Return Values

swinging

Boolean that is only true if the player's swarm is currently swinging.

Examples

```
// wait for the player to start swinging  
wait( !( player1.isSwinging() ) );
```

player.isTeleporting

Returns if the player's swarm is currently teleporting (see [player.teleportTo](#)).

Syntax

```
teleporting player.isTeleporting ();
```

Return Values

teleporting

Boolean that is only true if the player's swarm is currently teleporting.

Examples

```
// wait for the player to finish teleporting  
wait( player1.isTeleporting() );
```


player.kill

Causes the player to die and play its standard death animation.

Neither the emergency shield, the spawn protection nor the script invincibility (see [player.setInvincibility](#)) can prevent this from happening.

Syntax

```
player.kill ();
```

Examples

```
// kill the player  
player.kill();
```

player.moveTo

Causes the player's swarm to move to a given point in the world.

Deactivates the player's control (see [player.setPlayerControl](#)).

Syntax

```
player.moveTo ( x, y [, "wait"] );
```

Parameters

x, y

Numerical world coordinates for the swarm to move to.

"wait"

The script will wait until the swarm has reached its *x, y* target.

Examples

```
// move the player towards a script position called "finish"  
player1.moveTo( finish.getPos(), "wait" );
```

player.removeAllMarkers

Causes the player's swarm to lose all of its markers.

If the player's swarm is not alive or it does not have any marker, nothing will happen. Markers for bonuses ("Speed Boost" and "Dash Boost") will not be removed if the respective bonus is active.

Syntax

```
player.removeAllMarkers ();
```

Examples

```
// remove all markers from player1  
player1.removeAllMarkers();
```

player.removeMarker

Removes a given RGB color marker from the player's swarm.

If the player's swarm is not alive or it does not have the given marker, nothing will happen. Markers for bonuses ("Speed Boost" and "Dash Boost") cannot be removed if the bonus is active.

Syntax

```
player.removeMarker ( r, g, b );
```

Parameters

r, *g*, *b*

Marker's numerical RGB values, each ranging from 0.0 to 1.0.

RGB values for available bonuses are:

- **Speed Boost:** 0.2, 0.25, 1.0
- **Dash Boost:** 1.0, 0.75, 0.33

Examples

```
// remove the red keycard marker from player1  
create( "Var", "redKeycard", 1, 0, 0 );  
player1.removeMarker( redKeycard.get() );
```

player.setDeathHandling

Sets the player's death handling.

Every player starts with the death handling configured in the editor under "Player Death Handling". Multiplayer levels only support "respawn" and "off".

Syntax

```
player.setDeathHandling ( handling );
```

Parameters

handling

String containing one of the following options:

- **"loadCheckpoint"**: If the player's swarm dies, the last checkpoint will be loaded. If no checkpoint has been reached yet, the level will be restarted. Does not work in Multiplayer levels.
- **"respawn"**: If the player's swarm dies, a random player spawn will be selected to start the player's swarm's respawn process.
- **"endLevel"**: If the player's swarm dies, the level ends. Based on the level's win condition the game will be won or lost. Does not work in Multiplayer levels.
- **"off"**: If the player's swarm dies, nothing happens

Examples

```
// deactivate the player's death handling  
player1.setDeathHandling( "off" );
```

player.setDash

Blocks or unblocks the use of the player's dash ability.

Syntax

```
player.setDash ( dash );
```

Parameters

dash

Boolean to block (false) or unblock (true) the player's dash ability.

Examples

```
// block the player's dash ability  
player1.setDash( false );
```

player.setDroneLimit

Sets or deactivates a scripted drone limit for the player.

If there is a scripted drone limit, the player will not be able to get more drones by collecting enemy's drops or bonuses.

Syntax

```
player.setDroneLimit ( limit );  
player.setDroneLimit ( false );
```

Parameters

limit

Numerical drone limit ranging from 0 to 200.

false

Deactivates the scripted drone limit.

Examples

```
// set the drone limit to 0 for 3 seconds  
player1.setDroneLimit( 0 );  
wait( 180 );  
  
// deactivate the drone limit  
player1.setDroneLimit( false );
```

player.setInvincibility

Sets the player's script invincibility.

Does neither affect the player's swarm's spawn protection nor its emergency shield.

Syntax

```
player.setInvincibility ( invincibility );
```

Parameters

invincibility

Either a Boolean to activate (true) or deactivate (false) the player's script invincibility or a numerical amount of ticks the player will be invincible for.

Examples

```
// make the player invincible for 5 seconds  
player1.setInvincibility( 300 );
```


player.setLevel

Sets the player's swarm level.

Syntax

```
player.setLevel ( level );
```

Parameters

`level`

Numerical swarm level ranging from 0 to 4.

Examples

```
// set the player's swarm level to its maximum  
player1.setLevel( 4 );
```

player.setPlayerControl

Activates or deactivates the player's control.

If the player's control is deactivated, the player cannot move or use abilities anymore. This also triggers the black cinematic bars to appear/disappear (see [game.setBars](#)).

Syntax

```
player.setPlayerControl ( control );
```

Parameters

`control`

Boolean to activate (true) or deactivate (false) the player's control.

Examples

```
// move the player to a script position named "start" (deactivates the player's control)
player1.moveTo( start.getPos(), "wait" );

// activate the player's control again
player1.setPlayerControl( true )
```

player.setPos

Instantly sets the player's position to a given point in the world.

Syntax

```
player.setPos ( x, y );
```

Parameters

x, y

Numerical world coordinates for the player to be moved to. The coordinates have to be inside of at least one field wall and outside of every obstacle wall.

Examples

```
// instantly move the player to a script position named "start"  
player1.setPos( start.getPos() );
```

player.setScore

Sets the player's current score.

Does not work if the game has ended already (see [game.ended](#)).

Syntax

```
player.setScore ( score );
```

Parameters

score

Numerical amount of points the player's score will be set to, ranging from 0 to 999.999.999.

Examples

```
// reset the player's score  
player1.setScore( 0 );
```

player.setShoot

Blocks or unblocks the use of the player's shoot ability.

Syntax

```
player.setShoot ( shoot );
```

Parameters

`shoot`

Boolean to block (false) or unblock (true) the player's shoot ability.

Examples

```
// block the player's shoot ability  
player1.setShoot( false );
```

player.setSmoothStop

Activates or deactivates the player's swarm's "Smooth Stop" handling.

The handling causes the swarm to make a smooth stop if it is about to reach its move target (see [player.moveTo](#)).

Syntax

```
player.setSmoothStop ( smooth );
```

Parameters

smooth

Boolean to activate (true) or deactivate (false) the player's "Smooth Stop" handling.

Examples

```
// activate the player's "Smooth Stop" handling  
player1.setSmoothStop( true );
```

player.setTeam

Adds the player to the given team.

The player will automatically be removed from his/her last team.

Syntax

```
player.setTeam ( team );
```

Parameters

`team`

String containing the player's new team's script name.

Examples

```
// move player1 to the second team  
player1.setTeam( game.getTeam( 1 ) );
```

player.shoot

Causes the player's swarm to shoot for one tick.

The player's swarm needs to be alive and must not be dashing.

Syntax

```
player.shoot ();
```

Examples

```
// cause the player's swarm to shoot  
player1.shoot();
```


player.spawnSwarm

Spawns the player's swarm if it is not alive already.

Depending on the given parameters, the level's configured "Default Player Level" and a random and valid player spawn or a given swarm level and position will be used.

Syntax

```
player.spawnSwarm ( [level, x, y] );
```

Parameters

`level`

Numerical swarm level.

`x, y`

Numerical world coordinates for the swarm to spawn at. The coordinates have to be inside of at least one field wall and outside of every obstacle wall.

Examples

```
// spawn the player's swarm at a random and valid player spawn point  
player1.spawnSwarm( );
```

```
// spawn the player's swarm with maximum level at the coordinates of a script position named  
"start"  
player1.spawnSwarm( 4, start.getPos() );
```

player.teleportTo

Teleports the player's swarm to a given point in the world.

Syntax

```
player.teleportTo ( x, y [, "wait" ] );
```

Parameters

x, y

Numerical world coordinates for the swarm to teleport to.

"wait"

The script will wait until the swarm has finished its teleport.

Examples

```
// teleport the player's swarm to a script position named "start"  
player1.teleportTo( start.getPos() );
```

Player Spawn

playerSpawn.setState

Activates or deactivates a player spawn.

Players may only spawn on active player spawns. Player spawns are active by default.

Syntax

```
playerSpawn.setState ( active );
```

Parameters

active

Boolean to activate (true) or deactivate (false) the player spawn.

Examples

```
// deactivate a player spawn named "ps1"  
ps1.setState( false );
```

playerSpawn.setTeam

Assigns the player spawn to a given team.

Syntax

```
playerSpawn.setTeam ( team );
```

Parameters

team

String containing a team's script name or "any". If "any" is passed, the spawn point will be available for every team.

Examples

```
// assign a player spawn named "ps1" to the first team  
ps1.setTeam( game.getTeam( 0 ) );
```

```
// assign a player spawn named "ps1" to any team  
ps1.setTeam( "any" );
```

Screen

screen.print

Prints a given text for one tick.

Syntax

```
screen.print ( text, style, alignment, size, r, g, b, a, x, y [, z] );
```

Parameters

text

String containing the text to print.

style

String containing one of the following options:

- **"world"**: Interprets `x`, `y` [, `z`] as world coordinates.
- **"screen"**: Interprets `x`, `y` as screen coordinates, 0/0 is the screen's center.
- **"ul"**: Interprets `x`, `y` as screen coordinates, 0/0 is the screen's upper left corner.
- **"um"**: Interprets `x`, `y` as screen coordinates, 0/0 is the screen's top center.
- **"ur"**: Interprets `x`, `y` as screen coordinates, 0/0 is the screen's upper right corner.
- **"ml"**: Interprets `x`, `y` as screen coordinates, 0/0 is the screen's left center.
- **"mm"**: Like "screen".
- **"mr"**: Interprets `x`, `y` as screen coordinates, 0/0 is the screen's right center.
- **"ll"**: Interprets `x`, `y` as screen coordinates, 0/0 is the screen's lower left corner.
- **"lm"**: Interprets `x`, `y` as screen coordinates, 0/0 is the screen's bottom center.
- **"lr"**: Interprets `x`, `y` as screen coordinates, 0/0 is the screen's lower right corner.

alignment

String containing one of the following options: "left", "center", "right"

size

Text's numerical font size.

r, g, b

Text's numerical RGB values, each ranging from 0.0 to 1.0.

a

Text's numerical alpha value, reaching from 0.0 (invisible) to 1.0 (fully visible).

x, y [, z]

Text's numerical coordinates.

Examples

```
// show current death count stored in a variable named "deathCount" instead of current score
game.showScore( false );
while( true )
    // the given size and position is the exact same as used by the standard score display
    screen.print( deathCount.asString(), "um", "center", 37.5, 1, 1, 1, 1, 0, -37.5 );

    // wait for next tick and repeat endlessly
    wait();
end
```

```
// show current task stored in a variable named "task" instead of current time
game.showTime ( false );
while( true )
    // the given size and position is the exact same as used by the standard time display
    screen.print( task.asString(), "ul", "left", 37.5, 1, 1, 1, 1, 18.75, -37.5 );

    // wait for next tick and repeat endlessly
    wait();
end
```


Script Position

scriptPosition.getPos

Returns the script position's current world position.

Syntax

```
x, y scriptPosition.getPos ( [shiftX, shiftY] );
```

Parameters

shiftX, shiftY

Numerical coordinates that will be added to x, y to get a position relative to the script position, for example, a position to its right.

Return Values

x, y

Script position's current numerical world position.

Examples

```
// move the player to a script position named "start"  
player1.moveTo( start.getPos(), "wait" );
```

Team

team.addScoreBonus

Adds the given amount of points to the team's score bonus.

Does not work in Singleplayer levels. If the resulting score bonus is > 999.999.999, the score bonus is set to 999.999.999. If the resulting score bonus is < -999.999.999, the score bonus is set to -999.999.999. Can be used to subtract a value from the score bonus by passing a negative number. Does not work if the game has already ended (see [game.ended](#)).

Syntax

```
team.addScoreBonus ( bonus );
```

Parameters

[bonus](#)

Numerical amount of points to be added to the team's score bonus.

Examples

```
// add 1 to the first team's score bonus  
asCode( game.getTeam( 0 ) ).addScoreBonus( 1 );
```

team.getColor

Returns the team's color.

Syntax

```
r, g, b team.getColor ();
```

Return Values

r, g, b

Numerical RGB values for the team's color, each ranging from 0.0 to 1.0.

Examples

```
// add a marker to player1 matching its team's color  
player1.addMarker( asCode( player1.getTeam() ).getColor() );
```

team.getLobbySlotCount

Returns how many player slots this team had assigned by the lobby.

It does not matter if a slot was filled or not. In Singleplayer levels the lobby slot count is always 1.

Syntax

```
slots team.getLobbySlotCount ();
```

Return Values

slots

Numerical amount of lobby slots.

Examples

```
// check if the first team had exactly 1 lobby slot assigned to it
if( == ( asCode( game.getTeam( 0 ) ).getLobbySlotCount(), 1 ) )
    // the first team had exactly 1 lobby slot assigned
end
```

team.getName

Returns the team's name.

This is not the team's script name. It is the name that is shown on the scoreboard.

Syntax

```
name team.getName ( );
```

Return Values

name

String containing the team's name.

Examples

```
// announce that the second team is leading  
game.message( +( asCode( game.getTeam( 1 ) ).getName(), " is leading!" ) );
```

team.getPlayer

Returns the specified player's script name.

Syntax

```
name game.getPlayer ( id );
```

Parameters

`id`

Player's numerical id within the team, 0 for the first player.

Return Values

`name`

String containing the player's script name.

Examples

```
// save the script name of the first player in the second team  
create( "Var", "name", asCode( game.getTeam( 1 ) ).getPlayer( 0 ) );
```


team.getPlayerCount

Returns the team's current player count.

Syntax

```
count team.getPlayerCount ();
```

Return Values

count

Team's current numerical player count.

Examples

```
// announce the current player count for the third team  
game.message( +( asCode( game.getTeam( 2 ) ).getPlayerCount(), " players" ) );
```

team.getScore

Returns the team's current score.

The team's score is the sum of all its players' scores and the current team's score bonus.

Syntax

```
score team.getScore ();
```

Return Values

score

Team's current numerical score.

Examples

```
// check if the first team's score is higher a value saved in a variable named "minimum"
if( >( asCode( game.getTeam( 0 ) ).getScore(), minimum.get() ) )
    // the team's score is higher
end
```

team.getScoreBonus

Returns the team's current score bonus.

Does not work in Singleplayer levels.

Syntax

```
bonus team.getScoreBonus ( );
```

Return Values

bonus

Team's current numerical score bonus.

Examples

```
// check if the second team does have any score bonus
if( >( asCode( game.getTeam( 1 ) ).getScoreBonus(), 0 ) )
    // the second team does have a score bonus
end
```

team.setScoreBonus

Sets the team's score bonus.

Does not work in Singleplayer levels. Does not work if the game has already ended (see [game.ended](#)).

Syntax

```
team.setScoreBonus ( bonus );
```

Parameters

bonus

Numerical amount of points the team's score bonus will be set to, ranging from -999.999.999 to 999.999.999.

Examples

```
// reset the first team's score bonus  
asCode( game.getTeam( 0 ) ).setScoreBonus( 0 );
```

Variables

variable.asCode

Interprets the variable's value as code and executes it.

Syntax

```
execute variable.asCode ();
```

Return Values

execute

Code's return value after being executed by this function.

Examples

```
// save the first player in a variable named "king"
create( "Var", "king", game.getPlayer( 0 ) );

// ping the king's position
game.ping( king.asCode().getPos() );
```

variable.asScore

Returns a formatted score string for the variable's value interpreted as points.

Syntax

```
score variable.asScore ();
```

Return Values

score

Formatted score string like "100.000".

Examples

```
// announce the minimum score saved in a variable named "minimumScore"  
game.message( +( "Score ", minimumScore.asScore(), " to win!" ) );
```

variable.asString

Returns the variable's value interpreted as string.

Can be used to translate Booleans or numbers.

Syntax

```
string variable.asString ();
```

Return Values

string

Variable's value interpreted as string.

Examples

```
// save and announce the state of a player detector named "detector"  
create( "Var", "state", detector.get() );  
game.message( state.asString() );
```


variable.asTime

Returns a formatted time string for the variable's value interpreted as ticks.

Syntax

```
time variable.asTime ();
```

Return Values

time

Formatted time string like "02:30.000".

Examples

```
// announce the time penalty saved in a variable named "penalty"  
game.message( penalty.asTime() );
```

variable.get

Returns the variable's value.

Syntax

```
value variable.get ( [index] );
```

Parameters

`index`

Numerical index to access a single entry of an array.

Return Values

`value`

If no `index` is passed, this is the variable's complete content, each entry is separated by a comma. If an `index` is passed, this is only the array entry's value with the given `index`.

Examples

```
// save 3 numbers in an array
create( "Var", "array", 2, 4, 9 );

// announce the second number ("The second number is: 4")
game.message( +( "The second number is: ", array.get( 1 ) ) );

// announce the sum of all numbers by passing all of them to the + function ("The sum is: 15")
game.message( +( "The sum is: ", +( array.get() ) ) );
```

variable.getSize

Returns the variable's array size.

Syntax

```
size variable.getSize ();
```

Return Values

size

Variable's numerical array size. 0 for empty variables.

Examples

```
// save 3 numbers in an array
create( "Var", "array", 2, 4, 9 );

// announce the array's size ("The array size is: 3")
game.message( +( "The array size is: ", array.getSize() ) );
```

variable.insert

Inserts a given value at a given variable's array position and pushes back everything behind said position.

Syntax

```
inserted variable.insert ( index, value );
```

Parameters

`index`

Numerical array index for `value` to be inserted into the array. If `index` is greater than the variable's current size, empty entries will be added to the array.

`value`

Array's new value on position `index`.

Return Values

`inserted`

`value` that has just been inserted.

Examples

```
// save 2 numbers in an array
create( "Var", "array", 2, 9 );

// insert a new value in the middle (new content: 2, 4, 9)
array.insert( 1, 4 );

// insert a new value in the array's 5th position (new content: 2, 4, 9, , 8)
array.insert( 4, 8 );
```

variable.remove

Removes a given array entry and closes the resulting gap.

Syntax

```
result variable.remove ( index );
```

Parameters

`index`

Numerical entry index to be removed from the array.

Return Values

`result`

Variable's complete value after the call.

Examples

```
// save 3 numbers in an array
create( "Var", "array", 2, 4, 9 );

// remove the second value (new content: 2, 9)
array.remove( 1 );
```

variable.set

Sets the variable's complete value.

Syntax

```
result variable.set ( [value1 [...]] );
```

Parameters

`value1 [...]`

Dynamic amount of values. If no `value` is passed, the variable will be completely empty.

Return Values

`result`

Variable's complete value after the call.

Examples

```
// save 3 numbers in an array
create( "Var", "array", 2, 4, 9 );

// fill with 3 different numbers (new content: 8, 1, 7)
array.set( 8, 1, 7 );

// empty the complete array
array.set();
```

variable.setAt

Sets a given array entry to a given value.

Syntax

```
set variable.setAt ( index, value );
```

Parameters

index

Numerical array entry's index for *value* to be set at. If *index* is greater than the variable's current size, empty entries will be added to the array.

value

Array's new value on position *index*.

Return Values

inserted

value that has just been set.

Examples

```
// save 3 numbers in an array
create( "Var", "array", 2, 4, 9 );

// overwrite the middle number (new content: 2, 5, 9)
array.setAt( 1, 5 );

// set a new value at the array's 5th position (new content: 2, 5, 9, , 8)
array.setAt( 4, 8 );
```

variable.+=

Adds a dynamic amount of numbers or strings to the variable's current value.

This will change the variable's value. Does not work if the variable is an array.

Syntax

```
sum variable.+= ( value1 [...] );
```

Parameters

`value1 [...]`

Values that will be added to the variable's value.

Return Values

`sum`

If all `values` and the variable's value are numbers, this is the result of their addition. Otherwise this is a string containing every `value` and the variable's value interpreted as string and put together.

Examples

```
// create the variable "number" and set it to 1
create( "Var", "number", 1 );

// add 2, 3, 4 and 5 to number (result: 15)
number.+=( 2, 3, 4, 5 );
```

```
// create the variable "string" and set it to "This"
create( "Var", "string", "This" );

// add more text (result: "This is 1 string.")
string.+=( " is ", 1, " string." );
```


variable.-=

Subtracts a given number from the variable's value.

This will change the variable's value. Does not work if the variable is an array. Does not work if the variable's value is not a number.

Syntax

```
result variable.-= ( value1 );
```

Parameters

`value1`

Numerical value that will be subtracted from the variable's value.

Return Values

`result`

Subtraction's numerical result of the variable's value minus `value1`.

Examples

```
// create the variable "number" and set it to 10
create( "Var", "number", 10 );

// subtract 4 (result: 6)
number.-=( 4 );
```

variable.=*

Multiplies the variable's value with a given number.

This will change the variable's value. Does not work if the variable is an array. Does not work if the variable's value is not a number.

Syntax

```
product variable.* ( factor1 );
```

Parameters

`factor1`

Numerical value that will be multiplied with the variable's value.

Return Values

`product`

Multiplication's numerical result.

Examples

```
// create the variable "number" and set it to 10
create( "Var", "number", 10 );

// multiply with 5 (result: 50)
number.*=( 5 );
```

variable./=

Divides the variable's value by a given number.

This will change the variable's value. Does not work if the variable is an array. Does not work if the variable's value is not a number.

Syntax

```
quotient variable.- ( divisor );
```

Parameters

`divisor`

Division's divisor the variable's value will be divided by.

Return Values

`quotient`

Division's numerical result of the variable's value divided by `divisor`.

Examples

```
// create the variable "number" and set it to 10
create( "Var", "number", 10 );

// divide it by 2 (result: 5)
number./=( 2 );
```

Event Handling

This section contains a list of events and their parameters that are required for [game.getEvent](#) and [game.getEventValue](#).

collision

Posted whenever a collision between one of the following pairings occurs: Player / player, player / enemy, player / shot, player / hostile drone, enemy / drone or enemy / enemy (enemy / enemy collision events need to be activated with [game.setEvECollisionEvents](#) first).

Every collision will cause 2 events to be generated with mirrored pairings.

In case of destruction, the referenced object may or may not still exist (undefined). It is advised to use names of referenced objects for identification only.

Syntax

```
event game.getEvent ( "collision" [, filterName1, filterValue1 [...]] );
```

Parameters

`filterName1, filterValue1 [...]`

Pair containing a string for the filter's name and its value. The value's type depends on the given filter. A dynamic amount of filters is supported, no filter is required.

The following filter names are supported for the collision event:

- **"objectAType"**: The associated value is a string containing object A's type. May be "player", "enemy", "drone" or "shot".
- **"objectAName"**: The associated value is a string containing object A's name. If object A is a drone, it references the player. If object A is a shot, it references the enemy that fired the shot.
- **"objectAPos"**: The associated value contains object A's numerical 2d world coordinates.
- **"objectADamaged"**: The associated value is a Boolean that is only true if object A has been damaged, for example whenever a player's swarm has been hit while it was not shielded.
- **"objectADestroyed"**: The associated value is a Boolean that is only true if object A has been destroyed by the collision.
- **"objectAPreset"**: The associated value is a string containing a preset's name. If object A is an enemy, it is the name of the preset the enemy is based on. If object A is a shot, it is the name of the preset the enemy that fired the shot was based on when it fired the shot.

Each filter name also exists for the second object of the collision, called object B. For example: "objectBType" and "objectBPos".

Return Values

event

Boolean that is only true if a collision event was found that matches every `filterName/filterValue` pair.

Examples

```
// check if a player collided with a shot fired by an enemy based on a preset called "Drainer"
if( game.getEvent( "collision", "objectAType", "player", "objectBType", "shot",
"objectBPreset", "Drainer" ) )
    // player1 collided
end
```

```
// check if a player, whose name is stored in a variable called "hunter", damaged another
player with a drone
if( game.getEvent( "collision", "objectAType", "drone", "objectAName", hunter.get(),
"objectBType", "player", "objectBDamaged", true ) )
    // the hunter damaged another player
end
```

enemyEvent

Posted by the enemy action management (Action: "Event") which is configured via enemy presets.

In case of destruction, the referenced enemy may or may not still exist (undefined). It is advised to use names of referenced enemies for identification only.

Syntax

```
event game.getEvent ( "enemyAction" [, filterName1, filterValue1 [...]] );
```

Parameters

`filterName1, filterValue1 [...]`

Pair containing a string for the filter's name and its value. The value's type depends on the given filter. A dynamic amount of filters is supported, no filter is required.

The following filter names are supported for the enemy's action event:

- **"identifier"**: The associated value is a string containing the identifier that has been configured in the enemy's action management.
- **"enemyName"**: The associated value is a string containing the enemy's name.
- **"enemyPos"**: The associated value contains the enemy's numerical 2d world coordinates.
- **"enemyPreset"**: The associated value is a string containing the name of the preset the enemy is based on.

Return Values

`event`

Boolean that is only true if an enemyAction event was found that matches every `filterName/filterValue` pair.

Examples

```
// check if the enemy named "target" detected that a player is close by
if( game.getEvent( "enemyAction", "identifier", "playerClose", "enemyName", "target" ) )
    // player is close by
end
```