# Warzone2100 JavaScript Scripting API

## 1 Introduction

Warzone2100 contains a scripting language for implementing AIs, campaigns and some of the game rules. It uses JavaScript, so you should become familiar with this language before proceeding with this document. A number of very good guides to JavaScript exist on the Internet.

The following hard-coded files exist for game rules that use this API:

**multiplay/skirmish/rules.js** Default game rules - base setup, starting research, winning and losing.

**multiplay/script/scavfact.js** Scavenger AI. This script is active if scavengers are.

For ordinary AI scripts, these are controlled using '.ai' files that are present in the 'multiplayer/skirmish' directory. Here is an example of an '.ai' file that defines an AI implemented using this API:

```
[AI]
name = "Semperfi JS"
js = semperfi.js
```

It references a '.js' JavaScript file that needs to be in the same directory as the '.ai' file.

The code in a javascript file is accessed through specially named functions called 'events'. These are defined below. An event is expected to carry out some computation then return immediately. The game is on hold while an event is processed, so do not do too many things at once, or else the player will experience stuttering.

All global variables are saved when the game is saved. However, do not try to keep JavaScript objects that are returned from API functions defined here around in the global scope. They are not meant to be used this way, and bad things may happen. If you need to keep static arrays around, it is better to keep them locally defined to a function, as they will then not be saved and loaded.

One error that it is easy to make upon initially learning JavaScript and using this API, is to try to use the 'for (... in ...)' construct to iterate over an array of objects. This does not work! Instead, use code like the following:

```
var droidlist = enumDroid(me, DROID_CONSTRUCT);
for (var i = 0; i < droidlist.length; i++)
{
        var droid = droidlist[i];
        ...
}
```

The above code gets a list of all your construction droids, and iterates over them one by one.

The droid object that you receive here has multiple properties that can be accessed to learn more about it. These propertie are read-only, and cannot be changed. In fact, objects that you get are just a copies of game state, and do not give any access to changing the game state itself.

Any value written in ALL_CAPS_WITH_UNDERSCORES are enums, special read-only constants defined by the game.

## 1.1 Challenges

Challenges may load scripts as well, if a [scripts] section is present in the challenge file, and has the keys "extra" or "rules". The "extra" key sets up an additional script to be run during the challenge. The "rules" key sets up an alternative rules file, which means that the "rules.js" mentioned above is *not* run. In this case, you must implement your own rules for winning and losing, among other things. Here is an example of such a scripts section:

```
        [scripts]
        rules = towerdefense.js
```

You can also specify which AI scripts are to be run for each player. These must be given a path to the script, since you may sometimes want them from the AI directory ("multiplay/skirmish/") and sometimes from the challenge directory ("challenges/"). If you do not want an AI script to be loaded for a player (for example, if you want this player to be controlled from one of your challenge scripts), then you can give it the special value "null". Here is an example if a challenge player definition with its AI specified:

```
[player_2]
name = "Enemy"
team = 1
difficulty = "Medium"
position = 4
ai = multiplay/skirmish/semperfi.js
```

# 2 Common Objects

Some objects are described under the functions creating them. The following objects are produced by multiple functions and widely used throughout, so it is better to learn about them first.

Note the special term *game object* that is used in several places in this document. This refers to the results of any function returning a Structure, Droid, Feature or Base Object as described below. Some functions may take such objects as input parameters, in this case they may not take any other kind of object as input parameter instead.

## 2.1 Research

Describes a research item. The following properties are defined:

**power** Number of power points needed for starting the research.

**points** Number of resarch points needed to complete the research.

**started** A boolean saying whether or not this research has been started by current player or any of its allies.

**done** A boolean saying whether or not this research has been completed.

**name** A string containing the full name of the research.

**id** A string containing the index name of the research.

**type** The type will always be RESEARCH_DATA.

## 2.2 Structure

Describes a structure (building). It inherits all the properties of the base object (see below). In addition, the following properties are defined:

**status** The completeness status of the structure. It will be one of BEING_-BUILT and BUILT.

**type** The type will always be STRUCTURE.

**cost** What it would cost to build this structure. (3.2+ only)

**stattype** The stattype defines the type of structure. It will be one of HQ, FACTORY, POWER_GEN, RESOURCE_EXTRACTOR, LAS-SAT, DEFENSE, WALL, RESEARCH_LAB, REPAIR_FACILITY, CY-BORG_FACTORY, VTOL_FACTORY, REARM_PAD, SAT_UPLINK, GATE and COMMAND_CONTROL.

**modules** If the stattype is set to one of the factories, POWER_GEN or RESEARCH_LAB, then this property is set to the number of module upgrades it has.

**canHitAir** True if the structure has anti-air capabilities. (3.2+ only)

**canHitGround** True if the structure has anti-ground capabilities. (3.2+ only)

**isSensor** True if the structure has sensor ability. (3.2+ only)

**isCB** True if the structure has counter-battery ability. (3.2+ only)

**isRadarDetector** True if the structure has radar detector ability. (3.2+ only)

**range** Maximum range of its weapons. (3.2+ only)

**hasIndirect** One or more of the structure's weapons are indirect. (3.2+ only)

## 2.3 Feature

Describes a feature (a *game object* not owned by any player). It inherits all the properties of the base object (see below). In addition, the following properties are defined:

**type** It will always be FEATURE.

**stattype** The type of feature. Defined types are OIL_RESOURCE, OIL_DRUM and ARTIFACT.

**damageable** Can this feature be damaged?

## 2.4 Droid

Describes a droid. It inherits all the properties of the base object (see below). In addition, the following properties are defined:

**type** It will always be DROID.

**order** The current order of the droid. This is its plan. The following orders are defined:

**DORDER_ATTACK** Order a droid to attack something.

**DORDER_MOVE** Order a droid to move somewhere.

**DORDER_SCOUT** Order a droid to move somewhere and stop to attack anything on the way.

**DORDER_BUILD** Order a droid to build something.

**DORDER_HELPBUILD** Order a droid to help build something.

**DORDER_LINEBUILD** Order a droid to build something repeatedly in a line.

**DORDER_REPAIR** Order a droid to repair something.

**DORDER_RETREAT** Order a droid to retreat back to HQ.

**DORDER_PATROL** Order a droid to patrol.

**DORDER_DEMOLISH** Order a droid to demolish something.

**DORDER_EMBARK** Order a droid to embark on a transport.

**DORDER_DISEMBARK** Order a transport to disembark its units at the given position.

**DORDER_FIRESUPPORT** Order a droid to fire at whatever the target sensor is targeting. (3.2+ only)

**DORDER_STOP** Order a droid to stop whatever it is doing. (3.2+ only)

**DORDER_RTR** Order a droid to return for repairs. (3.2+ only)

**DORDER_RTB** Order a droid to return to base. (3.2+ only)

**DORDER_HOLD** Order a droid to hold its position. (3.2+ only)

**DORDER_REARM** Order a VTOL droid to rearm. If given a target, will go to specified rearm pad. If not, will go to nearest rearm pad. (3.2+ only)

**DORDER_OBSERVE** Order a droid to keep a target in sensor view. (3.2+ only)

**DORDER_RECOVER** Order a droid to pick up something. (3.2+ only)

**DORDER_RECYCLE** Order a droid to factory for recycling. (3.2+ only)

**action** The current action of the droid. This is how it intends to carry out its plan. The C++ code may change the action frequently as it tries to carry out its order. You never want to set the action directly, but it may be interesting to look at what it currently is.

**droidType** The droid's type. The following types are defined:

**DROID_CONSTRUCT** Trucks and cyborg constructors.

**DROID_WEAPON** Droids with weapon turrets, except cyborgs.

**DROID_PERSON** Non-cyborg two-legged units, like scavengers.

**DROID_REPAIR** Units with repair turret, including repair cyborgs.

**DROID_SENSOR** Units with sensor turret.

**DROID_ECM** Unit with ECM jammer turret.

**DROID_CYBORG** Cyborgs with weapons.

**DROID_TRANSPORTER** Cyborg transporter.

**DROID_SUPERTRANSPORTER** Droid transporter.

**DROID_COMMAND** Commanders.

**group** The group this droid is member of. This is a numerical ID. If not a member of any group, will be set to *null*.

**armed** The percentage of weapon capability that is fully armed. Will be *null* for droids other than VTOLs.

**experience** Amount of experience this droid has, based on damage it has dealt to enemies.

**cost** What it would cost to build the droid. (3.2+ only)

**isVTOL** True if the droid is VTOL. (3.2+ only)

**canHitAir** True if the droid has anti-air capabilities. (3.2+ only)

**canHitGround** True if the droid has anti-ground capabilities. (3.2+ only)

**isSensor** True if the droid has sensor ability. (3.2+ only)

**isCB** True if the droid has counter-battery ability. (3.2+ only)

**isRadarDetector** True if the droid has radar detector ability. (3.2+ only)

**hasIndirect** One or more of the droid's weapons are indirect. (3.2+ only)

**range** Maximum range of its weapons. (3.2+ only)

**body** The body component of the droid. (3.2+ only)

**propulsion** The propulsion component of the droid. (3.2+ only)

**weapons** The weapon components of the droid, as an array. Contains 'name', 'id', 'armed' percentage and 'lastFired' properties. (3.2+ only)

**cargoCapacity** Defined for transporters only: Total cargo capacity (number of items that will fit may depend on their size). (3.2+ only)

**cargoSpace** Defined for transporters only: Cargo capacity left. (3.2+ only)

**cargoCount** Defined for transporters only: Number of individual *items* in the cargo hold. (3.2+ only)

**cargoSize** The amount of cargo space the droid will take inside a transport. (3.2+ only)

## 2.5 Base Object

Describes a basic object. It will always be a droid, structure or feature, but sometimes the difference does not matter, and you can treat any of them simply as a basic object. The following properties are defined:

**type** It will be one of DROID, STRUCTURE or FEATURE.

**id** The unique ID of this object.

**x** X position of the object in tiles.

**y** Y position of the object in tiles.

**z** Z (height) position of the object in tiles.

**player** The player owning this object.

**selected** A boolean saying whether 'selectedPlayer' has selected this object.

**name** A user-friendly name for this object.

**health** Percentage that this object is damaged (where 100

**armour** Amount of armour points that protect against kinetic weapons.

**thermal** Amount of thermal protection that protect against heat based weapons.

**born** The game time at which this object was produced or came into the world. (3.2+ only)

### 2.6 Template

Describes a template type. Templates are droid designs that a player has created. This type is experimental and subject to change. The following properties are defined:

**id** The ID of this object.

**name** Name of the template.

**cost** The power cost of the template if put into production.

**droidType** The type of droid that would be created.

**body** The name of the body type.

**propulsion** The name of the propulsion type.

**brain** The name of the brain type.

**repair** The name of the repair type.

**ecm** The name of the ECM (electronic counter-measure) type.

**construct** The name of the construction type.

**weapons** An array of weapon names attached to this template.

## 3 Events

### 3.1 eventGameInit()

An event that is run once as the game is initialized. Not all game may have been properly initialized by this time, so use this only to initialize script state.

### 3.2 eventStartLevel()

An event that is run once the game has started and all game data has been loaded.

### 3.3 eventMissionTimeout()

An event that is run when the mission timer has run out.

### 3.4 eventVideoDone()

An event that is run when a video show stopped playing.

### 3.5 eventGameLoaded()

An event that is run when game is loaded from a saved game. There is usually no need to use this event.

### 3.6 eventGameSaving()

An event that is run before game is saved. There is usually no need to use this event.

### 3.7 eventGameSaved()

An event that is run after game is saved. There is usually no need to use this event.

### 3.8 eventTransporterLaunch(transport)

An event that is run when the mission transporter has been ordered to fly off.

### 3.9 eventTransporterArrived(transport)

An event that is run when the mission transporter has arrived at the map edge with reinforcements.

### 3.10 eventTransporterExit(transport)

An event that is run when the mission transporter has left the map.

### 3.11 eventTransporterDone(transport)

An event that is run when the mission transporter has no more reinforcements to deliver.

### 3.12 eventTransporterLanded(transport)

An event that is run when the mission transporter has landed with reinforcements.

### 3.13 eventPlayerLeft(player index)

An event that is run after a player has left the game.

### 3.14 eventCheatMode(entered)

Game entered or left cheat/debug mode. The entered parameter is true if cheat mode entered, false otherwise.

### 3.15 eventDroidIdle(droid)

A droid should be given new orders.

### 3.16 eventDroidBuilt(droid[, structure])

An event that is run every time a droid is built. The structure parameter is set if the droid was produced in a factory. It is not triggered for droid theft or gift (check *eventObjectTransfer* for that).

### 3.17 eventStructureBuilt(structure[, droid])

An event that is run every time a structure is produced. The droid parameter is set if the structure was built by a droid. It is not triggered for building theft (check *eventObjectTransfer* for that).

### 3.18 eventStructureReady(structure)

An event that is run every time a structure is ready to perform some special ability. It will only fire once, so if the time is not right, register your own timer to keep checking.

### 3.19 eventAttacked(victim, attacker)

An event that is run when an object belonging to the script's controlling player is attacked. The attacker parameter may be either a structure or a droid.

### 3.20 eventResearched(research, structure, player)

An event that is run whenever a new research is available. The structure parameter is set if the research comes from a research lab owned by the current player. If an ally does the research, the structure parameter will be set to null. The player parameter gives the player it is called for.

### 3.21 eventDestroyed(object)

An event that is run whenever an object is destroyed. Careful passing the parameter object around, since it is about to vanish!

### 3.22 eventPickup(feature, droid)

An event that is run whenever a feature is picked up. It is called for all players / scripts. Careful passing the parameter object around, since it is about to vanish! (3.2+ only)

### 3.23 eventObjectSeen(viewer, seen)

An event that is run sometimes when an object goes from not seen to seen. First parameter is *game object* doing the seeing, the next the game object being seen. This is event is throttled, and so is not called every time.

### 3.24 eventObjectTransfer(object, from)

An event that is run whenever an object is transferred between players, for example due to a Nexus Link weapon. The event is called after the object has been transferred, so the target player is in object.player. The event is called for both players.

### 3.25 eventChat(from, to, message)

An event that is run whenever a chat message is received. The *from* parameter is the player sending the chat message. For the moment, the *to* parameter is always the script player.

### 3.26 eventBeacon(x, y, from, to[, message])

An event that is run whenever a beacon message is received. The *from* parameter is the player sending the beacon. For the moment, the *to* parameter is always the script player. Message may be undefined.

### 3.27 eventBeaconRemoved(from, to)

An event that is run whenever a beacon message is removed. The *from* parameter is the player sending the beacon. For the moment, the *to* parameter is always the script player.

### 3.28 eventSelectionChanged(objects)

An event that is triggered whenever the host player selects one or more game objects. The *objects* parameter contains an array of the currently selected game objects. Keep in mind that the player may drag and drop select many units at once, select one unit specifically, or even add more selections to a current selection one at a time. This event will trigger once for each user action, not once for each selected or deselected object. If all selected game objects are deselected, *objects* will be empty.

### 3.29 eventGroupLoss(object, group id, new size)

An event that is run whenever a group becomes empty. Input parameter is the about to be killed object, the group's id, and the new group size.

### 3.30 eventArea¡label¿(droid)

An event that is run whenever a droid enters an area label. The area is then deactived. Call resetArea() to reactivate it. The name of the event is eventArea + the name of the label.

### 3.31 eventDesignCreated(template)

An event that is run whenever a new droid template is created. It is only run on the client of the player designing the template.

### 3.32 eventSyncRequest(req_id, x, y, obj_id, obj_id2)

An event that is called from a script and synchronized with all other scripts and hosts to prevent desync from happening. Sync requests must be carefully validated to prevent cheating!

## 4 Globals

**version** Current version of the game, set in *major.minor* format.

**selectedPlayer** The player ontrolled by the client on which the script runs.

**gameTime** The current game time. Updated before every invokation of a script.

**difficulty** The currently set campaign difficulty, or the current AI's difficulty setting. It will be one of EASY, MEDIUM, HARD or INSANE.

**mapName** The name of the current map.

**baseType** The area name of the map.

**baseType** The type of base that the game starts with. It will be one of CAMP_CLEAN, CAMP_BASE or CAMP_WALLS.

**alliancesType** The type of alliances permitted in this game. It will be one of NO_ALLIANCES, ALLIANCES or ALLIANCES_TEAMS.

**powerType** The power level set for this game.

**maxPlayers** The number of active players in this game.

**scavengers** Whether or not scavengers are activated in this game.

**mapWidth** Width of map in tiles.

**mapHeight** Height of map in tiles.

**scavengerPlayer** Index of scavenger player. (3.2+ only)

**isMultiplayer** If the current game is a online multiplayer game or not. (3.2+ only)

**me** The player the script is currently running as.

**scriptName** Base name of the script that is running.

**scriptPath** Base path of the script that is running.

**Stats** A sparse, read-only array containing rules information for game entity types. (For now only the highest level member attributes are documented here. Use the 'jsdebug' cheat to see them all.) These values are defined:

**Body** Droid bodies

**Sensor** Sensor turrets

**ECM** ECM (Electronic Counter-Measure) turrets

**Repair** Repair turrets (not used, incidentially, for repair centers)

**Construct** Constructor turrets (eg for trucks)

**Weapon** Weapon turrets

**WeaponClass** Defined weapon classes

**Building** Buildings

**Upgrades** A special array containing per-player rules information for game entity types, which can be written to in order to implement upgrades and other dynamic rules changes. Each item in the array contains a subset of the sparse array of rules information in the *Stats* global. These values are defined:

**Body** Droid bodies

**Sensor** Sensor turrets

**ECM** ECM (Electronic Counter-Measure) turrets

**Repair** Repair turrets (not used, incidentially, for repair centers)

**Construct** Constructor turrets (eg for trucks)

**Weapon** Weapon turrets

**Building** Buildings

**groupSizes** A sparse array of group sizes. If a group has never been used, the entry in this array will be undefined.

**playerData** An array of information about the players in a game. Each item in the array is an object containing the following variables: difficulty (see *difficulty* global constant), colour, position, isAI (3.2+ only), isHuman (3.2+ only), name (3.2+ only), and team.

**derrickPositions** An array of derrick starting positions on the current map. Each item in the array is an object containing the x and y variables for a derrick.

**startPositions** An array of player start positions on the current map. Each item in the array is an object containing the x and y variables for a player start position.

# 5 Functions

## 5.1 setTimer(function, milliseconds[, object])

Set a function to run repeated at some given time interval. The function to run is the first parameter, and it must be quoted, otherwise the function will be inlined. The second parameter is the interval, in milliseconds. A third, optional parameter can be a *game object* to pass to the timer function. If the *game object* dies, the timer stops running. The minimum number of milliseconds is 100, but such fast timers are strongly discouraged as they may deteriorate the game performance.

```
function conDroids()
{
    ... do stuff ...
}
// call conDroids every 4 seconds
setTimer("conDroids", 4000);
```

## 5.2 removeTimer(function)

Removes an existing timer. The first parameter is the function timer to remove, and its name must be quoted.

## 5.3 queue(function[, milliseconds[, object]])

Queues up a function to run at a later game frame. This is useful to prevent stuttering during the game, which can happen if too much script processing is done at once. The function to run is the first parameter, and it must be quoted, otherwise the function will be inlined. The second parameter is the delay in milliseconds, if it is omitted or 0, the function will be run at a later frame. A third optional parameter can be a *game object* to pass to the queued function. If the *game object* dies before the queued call runs, nothing happens.

## 5.4 include(file)

Includes another source code file at this point. You should generally only specify the filename, not try to specify its path, here.

## 5.5 getWeaponInfo(weapon id)

Return information about a particular weapon type. DEPRECATED - query the Stats object instead. (3.2+ only)

## 5.6 resetArea(label[, filter])

Reset the trigger on an area. Next time a unit enters the area, it will trigger an area event. Optionally add a filter on it in the second parameter, which can be a specific player to watch for, or ALL_PLAYERS by default. This is a fast operation of O(log n) algorithmic complexity. (3.2+ only)

## 5.7 enumLabels([filter])

Returns a string list of labels that exist for this map. The optional filter parameter can be used to only return labels of one specific type. (3.2+ only)

## 5.8 addLabel(object, label)

Add a label to a game object. If there already is a label by that name, it is overwritten. This is a fast operation of O(log n) algorithmic complexity. (3.2+ only)

## 5.9 removeLabel(label)

Remove a label from the game. Returns the number of labels removed, which should normally be either 1 (label found) or 0 (label not found). (3.2+ only)

## 5.10 getLabel(object)

Get a label string belonging to a game object. If the object has multiple labels, only the first label found will be returned. If the object has no labels, null is returned. This is a relatively slow operation of O(n) algorithmic complexity. (3.2+ only)

## 5.11 getObject(label — x, y — type, player, id)

Fetch something denoted by a label, a map position or its object ID. A label refers to an area, a position or a *game object* on the map defined using the map editor and stored together with the map. In this case, the only argument is a text label. The function returns an object that has a type variable defining what it is (in case this is unclear). This type will be one of DROID, STRUCTURE, FEATURE, AREA, GROUP or POSITION. The

AREA has defined 'x', 'y', 'x2', and 'y2', while POSITION has only defined 'x' and 'y'. The GROUP type has defined 'type' and 'id' of the group, which can be passed to enumGroup(). This is a fast operation of O(log n) algorithmic complexity. If the label is not found, an undefined value is returned. If whatever object the label should point at no longer exists, a null value is returned. You can also fetch a STRUCTURE or FEATURE type game object from a given map position (if any). This is a very fast operation of O(1) algorithmic complexity. Droids cannot be fetched in this manner, since they do not a unique placement on map tiles. Finally, you can fetch an object using its ID, in which case you need to pass its type, owner and unique object ID. This is an operation of O(n) algorithmic complexity. (3.2+ only)

## 5.12 enumBlips(player)

Return an array containing all the non-transient radar blips that the given player can see. This includes sensors revealed by radar detectors, as well as ECM jammers. It does not include units going out of view.

## 5.13 enumSelected()

Return an array containing all game objects currently selected by the host player. (3.2+ only)

## 5.14 enumGateways()

Return an array containing all the gateways on the current map. The array contains object with the properties x1, y1, x2 and y2. (3.2+ only)

## 5.15 enumTemplates(player)

Return an array containing all the buildable templates for the given player. (3.2+ only)

## 5.16 enumGroup(group)

Return an array containing all the members of a given group.

## 5.17 newGroup()

Allocate a new group. Returns its numerical ID. Deprecated since 3.2 - you should now use your own number scheme for groups.

## 5.18 activateStructure(structure, [target[, ability]])

Activate a special ability on a structure. Currently only works on the lassat. The lassat needs a target.

## 5.19 findResearch(research)

Return list of research items remaining to be researched for the given research item. (3.2+ only)

## 5.20 pursueResearch(lab, research)

Start researching the first available technology on the way to the given technology. First parameter is the structure to research in, which must be a research lab. The second parameter is the technology to pursue, as a text string as defined in "research.ini". The second parameter may also be an array of such strings. The first technology that has not yet been researched in that list will be pursued.

## 5.21 getResearch(research[, player])

Fetch information about a given technology item, given by a string that matches its definition in "research.ini". If not found, returns null.

## 5.22 enumResearch()

Returns an array of all research objects that are currently and immediately available for research.

## 5.23 componentAvailable([component type,] component name)

Checks whether a given component is available to the current player. The first argument is optional and deprecated.

## 5.24 addFeature(name, x, y)

Create and place a feature at the given x, y position. Will cause a desync in multiplayer. Returns the created game object on success, null otherwise. (3.2+ only)

## 5.25 addDroid(player, x, y, name, body, propulsion, reserved, reserved, turrets...)

Create and place a droid at the given x, y position as belonging to the given player, built with the given components. Currently does not support placing droids in multiplayer, doing so will cause a desync. Returns the created droid on success, otherwise returns null. Passing "" for reserved parameters is recommended.

## 5.26 makeTemplate(player, name, body, propulsion, reserved, turrets...)

Create a template (virtual droid) with the given components. Can be useful for calculating the cost of droids before putting them into production, for instance. Will fail and return null if template could not possibly be built using current research. (3.2+ only)

## 5.27 buildDroid(factory, name, body, propulsion, reserved, reserved, turrets...)

Start factory production of new droid with the given name, body, propulsion and turrets. The reserved parameter should be passed *null* for now. The components can be passed as ordinary strings, or as a list of strings. If passed as a list, the first available component in the list will be used. The second reserved parameter used to be a droid type. It is now unused and in 3.2+ should be passed "", while in 3.1 it should be the droid type to be built. Returns a boolean that is true if production was started.

## 5.28 enumStruct([player[, structure type[, looking player]]])

Returns an array of structure objects. If no parameters given, it will return all of the structures for the current player. The second parameter can be either

a string with the name of the structure type as defined in "structures.ini", or a stattype as defined in 2.2. The third parameter can be used to filter by visibility, the default is not to filter.

## 5.29 enumStructOffWorld([player[, structure type[, looking player]]])

Returns an array of structure objects in your base when on an off-world mission, NULL otherwise. If no parameters given, it will return all of the structures for the current player. The second parameter can be either a string with the name of the structure type as defined in "structures.ini", or a stattype as defined in 2.2. The third parameter can be used to filter by visibility, the default is not to filter.

## 5.30 enumFeature(player[, name])

Returns an array of all features seen by player of given name, as defined in "features.ini". If player is *ALL_PLAYERS*, it will return all features irrespective of visibility to any player. If name is empty, it will return any feature.

## 5.31 enumCargo(transport droid)

Returns an array of droid objects inside given transport. (3.2+ only)

## 5.32 enumDroid([player[, droid type[, looking player]]])

Returns an array of droid objects. If no parameters given, it will return all of the droids for the current player. The second, optional parameter is the name of the droid type. The third parameter can be used to filter by visibility - the default is not to filter.

## 5.33 dump(string...)

Output text to a debug file. (3.2+ only)

## 5.34 debug(string...)

Output text to the command line.

## 5.35 pickStructLocation(droid, structure type, x, y)

Pick a location for constructing a certain type of building near some given position. Returns an object containing "type" POSITION, and "x" and "y" values, if successful.

## 5.36 structureIdle(structure)

Is given structure idle?

## 5.37 removeStruct(structure)

Immediately remove the given structure from the map. Returns a boolean that is true on success. No special effects are applied. Deprecated since 3.2.

## 5.38 removeObject(game object[, special effects?])

Remove the given game object with special effects. Returns a boolean that is true on success. A second, optional boolean parameter specifies whether special effects are to be applied. (3.2+ only)

## 5.39 console(strings...)

Print text to the player console.

## 5.40 groupAddArea(group, x1, y1, x2, y2)

Add any droids inside the given area to the given group. (3.2+ only)

## 5.41 groupAddDroid(group, droid)

Add given droid to given group. Deprecated since 3.2 - use groupAdd() instead.

## 5.42 groupAdd(group, object)

Add given game object to the given group.

## 5.43 distBetweenTwoPoints(x1, y1, x2, y2)

Return distance between two points.

## 5.44 groupSize(group)

Return the number of droids currently in the given group. Note that you can use groupSizes[] instead.

## 5.45 droidCanReach(droid, x, y)

Return whether or not the given droid could possibly drive to the given position. Does not take player built blockades into account.

## 5.46 propulsionCanReach(propulsion, x1, y1, x2, y2)

Return true if a droid with a given propulsion is able to travel from (x1, y1) to (x2, y2). Does not take player built blockades into account. (3.2+ only)

## 5.47 terrainType(x, y)

Returns tile type of a given map tile, such as TER_WATER for water tiles or TER_CLIFFFACE for cliffs. Tile types regulate which units may pass through this tile. (3.2+ only)

## 5.48 orderDroid(droid, order)

Give a droid an order to do something. (3.2+ only)

## 5.49 orderDroidObj(droid, order, object)

Give a droid an order to do something to something.

## 5.50 orderDroidBuild(droid, order, structure type, x, y[, direction])

Give a droid an order to build someting at the given position. Returns true if allowed.

## 5.51 orderDroidLoc(droid, order, x, y)

Give a droid an order to do something at the given location.

## 5.52 setMissionTime(time)

Set mission countdown in seconds.

## 5.53 getMissionTime()

Get time remaining on mission countdown in seconds. (3.2+ only)

## 5.54 setTransporterExit(x, y, player)

Set the exit position for the mission transporter. (3.2+ only)

## 5.55 startTransporterEntry(x, y, player)

Set the entry position for the mission transporter, and make it start flying in reinforcements. If you want the camera to follow it in, use cameraTrack() on it. The transport needs to be set up with the mission droids, and the first transport found will be used. (3.2+ only)

## 5.56 setReinforcementTime(time)

Set time for reinforcements to arrive. If time is negative, the reinforcement GUI is removed and the timer stopped. Time is in seconds.

## 5.57 setStructureLimits(structure type, limit[, player])

Set build limits for a structure.

### 5.58 centreView(x, y)

Center the player's camera at the given position.

### 5.59 playSound(sound[, x, y, z])

Play a sound, optionally at a location.

### 5.60 gameOverMessage(won)

End game in victory or defeat.

### 5.61 completeResearch(research[, player])

Finish a research for the given player.

### 5.62 enableResearch(research[, player])

Enable a research for the given player, allowing it to be researched.

### 5.63 extraPowerTime(time, player)

Increase a player's power as if that player had power income equal to current income over the given amount of extra time. (3.2+ only)

### 5.64 setPower(power[, player])

Set a player's power directly. (Do not use this in an AI script.)

### 5.65 setPowerModifier(power[, player])

Set a player's power modifier percentage. (Do not use this in an AI script.) (3.2+ only)

### 5.66 enableStructure(structure type[, player])

The given structure type is made available to the given player. It will appear in the player's build list.

## 5.67 setTutorialMode(bool)

Sets a number of restrictions appropriate for tutorial if set to true.

## 5.68 setMiniMap(bool)

Turns visible minimap on or off in the GUI.

## 5.69 setDesign(bool)

Whether to allow player to design stuff.

## 5.70 enableTemplate(template name)

Enable a specific template (even if design is disabled).

## 5.71 setReticuleButton(id, filename, filenameHigh, tooltip, callback)

Add reticule button. id is which button to change, where zero is zero is the middle button, then going clockwise from the uppermost button. filename is button graphics and filenameHigh is for highlighting. The tooltip is the text you see when you mouse over the button. Finally, the callback is which scripting function to call. Hide and show the user interface for such changes to take effect. (3.2+ only)

## 5.72 showInterface()

Show user interface. (3.2+ only)

## 5.73 hideInterface(button type)

Hide user interface. (3.2+ only)

## 5.74 removeReticuleButton(button type)

Remove reticule button. DO NOT USE FOR ANYTHING.

## 5.75 applyLimitSet()

Mix user set limits with script set limits and defaults.

## 5.76 enableComponent(component, player)

The given component is made available for research for the given player.

## 5.77 makeComponentAvailable(component, player)

The given component is made available to the given player. This means the player can actually build designs with it.

## 5.78 allianceExistsBetween(player, player)

Returns true if an alliance exists between the two players, or they are the same player.

## 5.79 _(string)

Mark string for translation.

## 5.80 playerPower(player)

Return amount of power held by the given player.

## 5.81 queuedPower(player)

Return amount of power queued up for production by the given player. (3.2+ only)

## 5.82 isStructureAvailable(structure type[, player])

Returns true if given structure can be built. It checks both research and unit limits.

## 5.83 isVTOL(droid)

Returns true if given droid is a VTOL (not including transports).

## 5.84 hackGetObj(type, player, id)

Function to find and return a game object of DROID, FEATURE or STRUCTURE types, if it exists. Otherwise, it will return null. This function is deprecated by getObject(). (3.2+ only)

## 5.85 hackChangeMe(player)

Change the 'me' who owns this script to the given player. This needs to be run first in *eventGameInit* to make sure things do not get out of control.

## 5.86 receiveAllEvents(bool)

Make the current script receive all events, even those not meant for 'me'. (3.2+ only)

## 5.87 hackAssert(condition, message...)

Function to perform unit testing. It will throw a script error and a game assert. (3.2+ only)

## 5.88 objFromId(fake game object)

Broken function meant to make porting from the old scripting system easier. Do not use for new code. Instead, use labels.

## 5.89 setDroidExperience(droid, experience)

Set the amount of experience a droid has. Experience is read using floating point precision.

## 5.90 donateObject(object, to)

Donate a game object (currently restricted to droids) to another player. Returns true if donation was successful. May return false if this donation would push the receiving player over unit limits. (3.2+ only)

## 5.91 donatePower(amount, to)

Donate power to another player. Returns true. (3.2+ only)

## 5.92 safeDest(player, x, y)

Returns true if given player is safe from hostile fire at the given location, to the best of that player's map knowledge.

## 5.93 addStructure(structure type, player, x, y)

Create a structure on the given position. Returns the structure on success, null otherwise.

## 5.94 getStructureLimit(structure type[, player])

Returns build limits for a structure.

## 5.95 countStruct(structure type[, player])

Count the number of structures of a given type. The player parameter can be a specific player, ALL_PLAYERS, ALLIES or ENEMIES.

## 5.96 countDroid([droid type[, player]])

Count the number of droids that a given player has. Droid type must be either DROID_ANY, DROID_COMMAND or DROID_CONSTRUCT. The player parameter can be a specific player, ALL_PLAYERS, ALLIES or ENEMIES.

## 5.97 setNoGoArea(x1, y1, x2, y2, player)

Creates an area on the map on which nothing can be built. If player is zero, then landing lights are placed. If player is -1, then a limbo landing zone is created and limbo droids placed.

## 5.98 setScrollLimits(x1, y1, x2, y2)

Limit the scrollable area of the map to the given rectangle. (3.2+ only)

## 5.99 getScrollLimits()

Get the limits of the scrollable area of the map as an area object. (3.2+ only)

## 5.100 loadLevel(level name)

Load the level with the given name.

## 5.101 enumRange(x, y, range[, filter[, seen]])

Returns an array of game objects seen within range of given position that passes the optional filter which can be one of a player index, ALL_PLAYERS, ALLIES or ENEMIES. By default, filter is ALL_PLAYERS. Finally an optional parameter can specify whether only visible objects should be returned; by default only visible objects are returned. Calling this function is much faster than iterating over all game objects using other enum functions. (3.2+ only)

## 5.102 enumArea(¡x1, y1, x2, y2 — label¿[, filter[, seen]])

Returns an array of game objects seen within the given area that passes the optional filter which can be one of a player index, ALL_PLAYERS, ALLIES or ENEMIES. By default, filter is ALL_PLAYERS. Finally an optional parameter can specify whether only visible objects should be returned; by default only visible objects are returned. The label can either be actual positions or a label to an AREA. Calling this function is much faster than iterating over all game objects using other enum functions. (3.2+ only)

## 5.103 addBeacon(x, y, target player[, message])

Send a beacon message to target player. Target may also be *ALLIES*. Message is currently unused. Returns a boolean that is true on success. (3.2+ only)

## 5.104 removeBeacon(target player)

Remove a beacon message sent to target player. Target may also be *ALLIES*. Returns a boolean that is true on success. (3.2+ only)

### 5.105 chat(target player, message)

Send a message to target player. Target may also be *ALL_PLAYERS* or *ALLIES*. Returns a boolean that is true on success. (3.2+ only)

### 5.106 setAlliance(player1, player2, value)

Set alliance status between two players to either true or false. (3.2+ only)

### 5.107 setAssemblyPoint(structure, x, y)

Set the assembly point droids go to when built for the specified structure. (3.2+ only)

### 5.108 hackNetOff()

Turn off network transmissions. FIXME - find a better way.

### 5.109 hackNetOn()

Turn on network transmissions. FIXME - find a better way.

### 5.110 getDroidProduction(factory)

Return droid in production in given factory. Note that this droid is fully virtual, and should never be passed anywhere. (3.2+ only)

### 5.111 getDroidLimit([player[, unit type]])

Return maximum number of droids that this player can produce. This limit is usually fixed throughout a game and the same for all players. If no arguments are passed, returns general unit limit for the current player. If a second, unit type argument is passed, the limit for this unit type is returned, which may be different from the general unit limit (eg for commanders and construction droids). (3.2+ only)

### 5.112 getExperienceModifier(player)

Get the

### 5.113 setExperienceModifier(player, percent)

Set the

### 5.114 setDroidLimit(player, value[, droid type])

Set the maximum number of droids that this player can produce. If a third parameter is added, this is the droid type to limit. It can be DROID_ANY for droids in general, DROID_CONSTRUCT for constructors, or DROID_-COMMAND for commanders. (3.2+ only)

### 5.115 setCommanderLimit(player, value)

Set the maximum number of commanders that this player can produce. THIS FUNCTION IS DEPRECATED AND WILL BE REMOVED! (3.2+ only)

### 5.116 setConstructorLimit(player, value)

Set the maximum number of constructors that this player can produce. THIS FUNCTION IS DEPRECATED AND WILL BE REMOVED! (3.2+ only)

### 5.117 hackAddMessage(message, type, player, immediate)

See wzscript docs for info, to the extent any exist. (3.2+ only)

### 5.118 hackRemoveMessage(message, type, player)

See wzscript docs for info, to the extent any exist. (3.2+ only)

### 5.119 setSunPosition(x, y, z)

Move the position of the Sun, which in turn moves where shadows are cast. (3.2+ only)

### 5.120 setSunIntensity(ambient r, g, b, diffuse r, g, b, specular r, g, b)

Set the ambient, diffuse and specular colour intensities of the Sun lighting source. (3.2+ only)

### 5.121 setWeather(weather type)

Set the current weather. This should be one of WEATHER_RAIN, WEATHER_SNOW or WEATHER_CLEAR. (3.2+ only)

### 5.122 setSky(texture file, wind speed, skybox scale)

Change the skybox. (3.2+ only)

### 5.123 hackMarkTiles([label — x, y[, x2, y2]])

Mark the given tile(s) on the map. Either give a POSITION or AREA label, or a tile x, y position, or four positions for a square area. If no parameter is given, all marked tiles are cleared. (3.2+ only)

### 5.124 cameraSlide(x, y)

Slide the camera over to the given position on the map. (3.2+ only)

### 5.125 cameraZoom(z, speed)

Slide the camera to the given zoom distance. Normal camera zoom ranges between 500 and 5000. (3.2+ only)

### 5.126 cameraTrack(droid)

Make the camera follow the given droid object around. Pass in a null object to stop. (3.2+ only)

### 5.127 addSpotter(x, y, player, range, type, expiry)

Add an invisible viewer at a given position for given player that shows map in given range. *type* is zero for vision reveal, or one for radar reveal. The difference is that a radar reveal can be obstructed by ECM jammers. *expiry*, if non-zero, is the game time at which the spotter shall automatically be removed. The function returns a unique ID that can be used to remove the spotter with *removeSpotter*. (3.2+ only)

### 5.128 removeSpotter(id)

Remove a spotter given its unique ID. (3.2+ only)

### 5.129 syncRandom(limit)

Generate a synchronized random number in range 0...(limit - 1) that will be the same if this function is run on all network peers in the same game frame. If it is called on just one peer (such as would be the case for AIs, for instance), then game sync will break. (3.2+ only)

### 5.130 syncRequest(req_id, x, y[, obj[, obj2]])

Generate a synchronized event request that is sent over the network to all clients and executed simultaneously. Must be caught in an eventSyncRequest() function. All sync requests must be validated when received, and always take care only to define sync requests that can be validated against cheating. (3.2+ only)

### 5.131 replaceTexture(old_filename, new_filename)

Replace one texture with another. This can be used to for example give buildings on a specific tileset different looks, or to add variety to the looks of droids in campaign missions. (3.2+ only)

# 6 Gotchas / Bugs

## 6.1 Case sensitivity

Due to a bug that is not so easy to fix in the short term, variables defined in global must be case insensitive. Otherwise, they may collide on savegame loading. This is only for variables defined in your script. There is no need to maintain case insensitivity in regards to variables defined in global by the game itself, such as 'FACTORY' (you can safely define your own 'factory' variable) – the only exception to this is 'me'.

## 6.2 Global objects

You must never put a *game object*, such as a droid or a structure, on global. You only get a snapshot of their state, the state is not updated, and they are not removed when they die. Trying to store them globally then using them later will fail.

All variables stored on global (in global scope) are stored when the game is saved, and restored when it is loaded. However, this may not work properly for complex objects. Basic arrays and basic types are supported, but it is generally not recommended to put objects on global, even though simple ones may work. Since the game can't be saved while a function is running, you don't need to worry about local variables.

Const definitions are not stored in savegames, and are therefore recommended over variables to hold information that does not change.

## 6.3 Object states

Most object states that are changed from the scripts are not in fact changed, but queued up to be synchronized among clients. This means that you cannot check the state later in the same script invokation and expect it to have changed. This includes such things as giving orders to droids, setting production to structures, and so on. Instead, if for example you want to mark droids that have been ordered to do something, you can mark them by adding a custom property. Note that this property will not be remembered when it goes out of scope.

## 6.4 Early research

You cannot set research topics for research labs directly from eventStartLevel. Instead, queue up a function call to set it at some later frame.

## 6.5 Cyborg construction

Cyborg components are inter-linked, and cannot be passed in as lists as you can with ordinary droids, even though they might look that way.

# 7 libcampaign.js documentation

`libcampaign.js` is a JavaScript library supplied with the game, which contains reusable code for campaign scenarios. It is designed to make scenario development as high-level and declarative as possible. It also contains a few simple convenient wrappers. Public API functions of `libcampaign.js` are prefixed with '`cam`'. To use `libcampaign.js`, add the following include into your scenario code:

```
include("script/campaign/libcampaign.js");
```

Also, most of the `libcampaign.js` features require some of the game events handled by the library. Transparent JavaScript pre-hooks are therefore injected into your global event handlers upon include. For example, if `camSetArtifacts()` was called to let `libcampaign.js` manage scenario artifacts, then `eventPickup()` will be first handled by the library, and only then your handler will be called, if any. All of this happens automagically and does not normally require your attention.

## 7.1 camDef(something)

Returns false if something is JavaScript-undefined, true otherwise.

## 7.2 camIsString(something)

Returns true if something is a string, false otherwise.

## 7.3 camRand(max)

A non-synchronous random integer in range [0, max - 1].

## 7.4 camSafeRemoveObject(obj[, special effects?])

Remove a game object (by value or label) if it exists, do nothing otherwise.

## 7.5 camMakePos(x, y — label — object)

Make a POSITION-like object, unless already done. Often useful for making functions that would accept positions in both xx,yy and x:xx,y:yy forms. Also accepts labels. If label of AREA is given, returns a random spot in the area. If an existing object or label of such is given, returns a safe JavaScript object containing its x, y and id.

## 7.6 camDist(x1, y1, x2, y2 — pos1, x2, y2 — x1, y1, pos2 — pos1, pos2)

A wrapper for distBetweenTwoPoints.

## 7.7 camPlayerMatchesFilter(player, filter)

A function to handle player filters in a way similar to how JS API functions (eg. `enumDroid(filter, ...)`) handle them.

## 7.8 camMakeGroup(what, filter)

Make a new group out of array of droids, single game object, or label string, with fuzzy auto-detection of argument type. Only droids would be added to the group. **filter** can be one of a player index, ALL_PLAYERS, ALLIES or ENEMIES.

## 7.9 camMarkTiles(label)

Mark area on the map by label, but only if debug mode is enabled. Otherwise, remember what to mark when it will be.

## 7.10 camDebug(string...)

Pretty debug prints - a wrapper around `debug()`. Prints a "DEBUG" prefix, then caller function name, then argument strings. Only use this function to indicate actual bugs in the scenario script, because game shouldn't print things when nothing is broken. For easier debugging, use `camTrace()`.

## 7.11 camDebugOnce(string...)

Same as `camDebug()`, but prints each message only once during script lifetime.

## 7.12 camTrace(string...)

Same as `camDebug()`, but only warns in cheat mode. Prefixed with "TRACE". It's safe and natural to keep `camTrace()` calls in your code for easier debugging.

## 7.13 camSetArtifacts(artifacts)

Tell `libcampaign.js` to manage a certain set of artifacts. The argument is a JavaScript map from object labels to artifact description. If the label points to a game object, artifact will be placed when this object is destroyed; if the label is a position, the artifact will be placed instantly. Artifact description is a JavaScript object with the following fields:

**tech** The technology to grant when the artifact is recovered.

On **let me win** cheat, all technologies stored in the artifacts managed by this function are automatically granted.

## 7.14 camAllArtifactsPickedUp()

Returns true if all artifacts managed by `libcampaign.js` were picked up.

## 7.15 camSetEnemyBases(bases)

Tell `libcampaign.js` to manage a certain set of enemy bases. Management assumes auto-cleanup of leftovers on destruction, and also counting how many

bases have been successfully destroyed by the player. The argument is a JavaScript map from group labels to base descriptions. Each label points to a group of vital base structures. Once this group is eradicated, the base is considered to be destroyed. Base description is a JavaScript object with the following fields:

**area** An area label to clean up features in once base is destroyed.

## 7.16 camAllEnemyBasesEliminated()

Returns true if all enemy bases managed by `libcampaign.js` are destroyed.

## 7.17 camManageGroup(group, order, data)

Tell `libcampaign.js` to manage a certain group. The group would be permanently managed depending on the high-level orders given. For each order, data parameter is a JavaScript object that controls different aspects of behavior. The order parameter is one of:

**CAM_ORDER_ATTACK** Pursue human player, preferably around the given position. The following optional data object fields are available:

**pos** Position to attack.

**fallback** Position to retreat.

**morale** An integer from 1 to 100. If that high percentage of the original group dies, fall back to the fallback position. If new droids are added to the group, it can recover and attack again.

**CAM_ORDER_DEFEND** Protect the given position. If too far, retreat back there ignoring fire. The following optional data object fields are available:

**pos** Position to defend.

## 7.18 camStopManagingGroup(group)

Tell `libcampaign.js` to stop managing a certain group.

## 7.19 camOrderToString(order)

Print campaign order as string, useful for debugging.

## 7.20 camSetFactories(factories)

Tell `libcampaign.js` to manage a certain set of enemy factories. Management assumes producing droids, packing them into groups and executing orders once the group becomes large-enough. The argument is a JavaScript map from group labels to factory descriptions. Each label points to a factory object. Factory description is a JavaScript object with the following fields:

**assembly** A rally point position label, where the group would gather.

**groupSize** Number of droids to produce before executing the order.

**order** An order to execute for every group produced in the factory. Same as the order parameter for `camManageGroup()`.

**data** Order data. Same as the data parameter for `camManageGroup()`.

**templates** List of droid templates to produce in the factory. Each template is a JavaScript object with the following fields:

- **body** Body stat name.
- **prop** Propulsion stat name.
- **weap** Weapon stat name. Only single-turret droids are currently supported.

Factories won't start production immediately; call `camEnableFactory()` to turn them on when necessary.

## 7.21 camEnableFactory(factoryLabel)

Enable a managed factory by the given label. Once the factory is enabled, it starts producing units and executing orders as given.

## 7.22 camNextLevel(nextLevel)

A wrapper around `loadLevel()`. Remembers to give bonus power for completing the mission faster - 25 percent more than what you could have received by milking this time limit down.