

A guide to scripting with Phasor

Made By: Oxide aka Genocide, urbanyoung.

Version 01.00.10.057

I've removed an introduction I made for Lua from this guide. The main reason for doing so is that it wasn't very good, and people have told me they would like to make one.

Before reading this guide I recommend you take a look at [A guide to Halo's Player and Object Management](#) as it explains stuff like memory ids, player ids, object ids, memory addresses, tags and offsets.

Listed below are functions that Phasor exports, or gives scripts access to. If a function fails to execute, nil is returned. So, check against nil for validity.

General functions:

- `hprintf(message)` – This function takes a string of text and prints it to the server console. If processing an rcon command the text is also forwarded to the player using rcon.
- `resolveplayer(player)` – This function resolves a player's memory id to their rcon (or player) id. The input is 0 based and the output is 1 based.
- `rresolveplayer(player)` – This function resolves a player's rcon (or player) id into their memory id. The input is 1 based and output is 0 based.
- `getobject(m_objId)` – This function resolves an object id to the object's memory address. This address can be used in subsequent memory operations (see below).
- `getplayer(player)` – This function resolves a player's memory id to their memory address. This address can be used in subsequent memory operations (see below).
- `changeteam(player)` – This function changes the specified player's team.
- `kill(player)` – This function kills the specified player.
- `applycamo(player, duration)` – This function makes a player invisible (camo) for the specified duration. 0 represents an infinite duration. The effect wears off when the player dies.
- `getteam(player)` – This function returns a player's team.
- `lookuptag(tagType, tag)` – This function requires both the tag type (ie weap) and the tag name. It returns the map id of the tag.
- `setspeed(player, speed)` – This function changes the specified player's movement speed. The default speed is 1.0. The greater the speed the more apparent lag is.
- `say(msg)` - This function sends the specified message to the server. Note: only ASCII characters are supported due to Lua's lack of unicode support.

- `privatesay(player, msg)` – This function sends a server message to the specific player. Note: only ASCII characters are supported due to Lua's (lack of) unicode support.
- `registertimer(delay, callback, opt: userdata1, userdata2 ...)` – This function can be used to create timers. The delay is in milliseconds so 1000 = 1 second. When the time runs out the function specified by callback is called. This callback function should return 1 to reset (reuse) the timer or 0 to remove it. You can use userdata to specify any custom data you wish but it must only be of type string, number, boolean. The return value is the id of the created timer. The callback function should be declared with the following arguments (in this order): id, count, opt: userdata1, userdata2, ... Count is the number of times the timer has been called.
- `removetimer(id)` – This function is used to remove (stop) a timer previously registered with `registertimer`.
- `getrandomnumber(min, max)` – This function returns a psuedo random number between the two specified limits.
- `getname(player)` – This function returns the name of the specified player. The name is returned in ASCII format, not unicode.
- `gethash(player)` – This function returns the cdkey hash of the specified player.
- `getteamsizeteam)` – This function returns the number of players on the specified team.
- `svcmd(cmd)` – This function executes the specified server command. Note: Any player indices should be of the PLAYER id (see `resolveplayer`).
- `movobjcoords(m_objectId, x, y, z)` – This function moves an object to the specified coordinates.
- `movobjname(m_objectId, name)` – This function moves an object to the coordinates specified by the named location.
- `updateammo(m_weaponId)` – This function sends a packet which causes an ammo changes made to the specified weapon sync correctly.
- `getplayerobjectid(player)` – This function returns a player's object id, should be checked against 0xffffffff to check if it exists.
- `getobjectcoords(m_objectId)` – This function returns the current coordinates of the specified object. If the object is a player and is in a vehicle, the coordinates of the vehicle is returned. Example: `local x,y,z = getobjectcoords(objId)`

- `isadmin(player)` – This function checks if a player is an admin. Returns true if they are or false if not.
- `createobject(tagType, tag, parentId, respawnTime, bRecycle, x, y, z)` – This function creates an object and returns the new object's id. `tagType` is the type of tag, ie "weap", `tag` is the absolute name of the tag, ie "weapons\\pistol\\pistol", `parentId` is the object id of the parent, `respawnTime` specifies how many seconds inactive Phasor should wait before respawning the object (0 for no respawn, -1 for gametype value). `bRecycle` specifies whether the object should respawn or be destroyed once the `respawnTime` runs out. `x,y,z` are the coordinates the object spawns at. Note: When spawning items with an item collection tag (weapons, nades, powerups) `bRecycle` is ignored and `respawnTime` is considered to be destruction time (time until object is destroyed).
- `destroyobject(m_objectId)` – This function destroys the object specified by `m_objectId`. Care should be taken to make sure the passed id is valid, otherwise the server may crash.
- `assignweapon(player, m_objectId)` – This function gives a player the weapon specified by `m_objectId`. The given weapon becomes their active weapon. This function only works if there is room for the weapon to go. (max 4 weapons).
- `entervehicle(player, m_vehicleId, seat)` – This function forces a player into the specified vehicle. Care should be taken that a player is not currently in a vehicle. `Seat` specifies what seat the player will enter: 0 (driver), 1 (passenger), 2 (gunner)
- `exitvehicle(player)` – This function forces a player to leave their current vehicle.
- `getprofilepath()` – This function returns the location of the server's data path (that is, where the banlist is stored).

Tokenization Functions:

The below functions are used for manipulating strings. You don't need to use these if you don't want to. Lua provides you with the tools to make this yourself. However, I'm familiar with C++ and as such used it to save time. The cmd tokenization functions tokenize at a space, however all spaces are ignored within a "" block.

- `gettokencount(source, delim)` – This function gets the number of times the delimiter appears in a source string.
- `gettoken(source, delim, token)` – This function returns the specified token within a string. *Note: Tokens start at 0.*
- `getcmdtokencount(source)` – This function returns the number of command tokens within the specified string.

- `getcmdtoken(source, token)` – This function returns the specified command token.

Note: Tokens start at 0.

Memory Operations:

The below functions are used to manipulate memory. They all require a base address and an offset. The data is read/written from base + offset.

- `readbit(data, data_offset, bit_offset)` – This function returns the bit at the specified memory location and offset. A bit is either 0 or 1. `bit_offset` specifies which bit of the specified byte is to be read, must be between 0 and 7.
- `readbyte(data, offset)` – This function returns the byte at the specified memory location. A byte is 8 bits.
- `readword(data, offset)` – This function returns the word at the specified memory location. A word is 16 bits.
- `readdword(data, offset)` – This function returns the dword at the specified memory location. A dword is 32 bits.
- `readfloat(data, offset)` – This function returns the float at the specified memory location. A float is 32 bits. The difference between a float and a dword is how the data is processed. Floats allow for decimalized numbers (ie 1.2) whereas a dword does not.
- `writebit(data_addr, data_offset, bit_offset, bit)` – This function writes the specified bit to the specified address.
- `writebyte(data_addr, offset, data)` – This function writes the specified data to the specified address.
- `writeword(data_addr, offset, data)` – This function writes the specified data to the specified address.
- `writedword(data_addr, offset, data)` – This function writes the specified data to the specified address.
- `writefloat(data_addr, offset, data)` – This function writes the specified data to the specified address.

Phasor makes heavy use of events that happen within game. Scripts get notified when the below events happen. You should note that when Phasor notifies scripts of events it calls the function whose name matches that of the event. You should also note that only functions that say so should have a return value.

1. OnScriptLoad(process) – This is called when a script has been successfully loaded. The parameter is the process id of the current process.
2. OnScriptUnload() – This is called when a script is being unloaded.
3. OnNewGame(map) – This is called when a new game starts. The parameter is the name of the map that is running.
4. OnGameEnd(mode) – This is called 3 times as the game ends. Possible mode values:
 - 1 - The game just ended (F1 scorecard is being displayed).
 - 2 – The post game scorecard is displayed.
 - 3 - Players can quit via the post game scorecard.
5. OnServerChat(player, chattype, message) – This is called when there is any chat within the server. Note: message is in ASCII. Possible chattype values:
 - 1 – All chat.
 - 2 – Team chat.
 - 3 – Vehicle chat.

Return value: This represents whether or not the message should be sent to the rest of the server. 1 allows the message, 0 blocks it.

6. OnServerCommand(player, command) – This is called when there's a server command to process. 'player' represents the player executing the command, if it is - 1 then the command is executed through the server console.

Return value: This represents whether or not to continue the processing of the command. 1 allows it, 0 blocks it.

7. OnTeamDecision(cur_team) – This is called when a player is being assigned a team. This function being called does not guarantee that the player will join the server.

Return value: This represents the team the player should be on.

8. OnPlayerJoin(player, team) – This is called when a player successfully joins the server. If a join notification is received, a leave notification will also be received as the player leaves (see OnPlayerLeave).
9. OnPlayerLeave(player, team) – This is called when a player leaves the server.
10. OnPlayerKill(killer, victim, mode) – This is called when a player is killed. The victim is always a valid player index, however killer may not be depending on the value of mode. Possible mode values:

- 0 – The player was killed by the server.
- 1 – The player was killed by falling.
- 2 – The player was killed by the guardians.
- 3 – The player was killed by a vehicle.
- 4 – The player was killed by another. *Killer is valid.*
- 5 – The player was betrayed by another. *Killer is valid.*
- 6 – The player committed suicide.

11. OnKillMultiplier(player, multiplier) – This is called when a player receives a kill multiplier (ie double kill). Possible multiplier values:

- 0x07 – Double kill
- 0x09 – Triple kill
- 0x0A – Killtacular
- 0x0B – Killing spree
- 0x0C – Running riot
- 0x0D – Betrayed
- 0x0E – Killtacular with score
- 0x0F – Triple kill with score
- 0x10 – Double kill with score
- 0x11 – Running riot with score
- 0x12 – Killing spree with score.

Please note that the function isn't called with all these values, they are just the possible values. Values with the score are generally used in Slayer.

12. OnPlayerSpawn (player, m_objectId) – This is called just before clients (players) are notified of a player spawning. m_objectId is the specified player's new object id. All object data that is modified here will sync.

13. OnPlayerSpawnEnd (player, m_objectId) – This is called just after clients (players) are notified of a player spawning. m_objectId is the specified player's new object id. Object data that is modified here may not sync.

14. OnWeaponReload(player, weapon) – This is called as a player attempts to reload, weapon is the object id of the weapon they're reloading.

Return value: This represents whether or not to allow the reload. 1 allows it, 0 blocks it.

15. `OnTeamChange(relevant, player, team, dest_team)` – This is called when a player is changing team. If relevant is 0 then the return value is ignored and the team change cannot be stopped.

Return values: The return value is only considered if 'relevant' is 1 and indicates whether or not the change should be allowed. 1 allows the change, 0 blocks it. Be careful with this function and multiple scripts as relevant can still be 1, even if the return value isn't considered.

16. `OnObjectInteraction(player, m_ObjectId, tagType, tagName)` – This is called when a player interacts with an object. An interaction is defined as either standing over the object or attempting to use it (pickup weapon, grenades etc). `m_ObjectId` is the id of the object that the player's interacting with. You can get information about what kind of object it is with the `tagType` and `tagName` parameters. For a list of tag types/tag names see [A guide to Halo's Player and Object Management](#).

Return value: This represents whether or not to allow the interaction. 1 allows it, 0 blocks it.

17. `OnVehicleEntry(relevant, player, vehicleId, vehicle_tag, seat)` – This is called when a player attempts to enter a vehicle. relevant specifies whether the vehicle entry can be stopped (using the script function `entervehicle()` is not stoppable).

Return value: This represents whether or not they're allowed to enter. 1 allows it, 0 blocks it.

18. `OnVehicleEject(player, forceEject)` – This is called when a player is about to leave a vehicle. `forceEject` specifies whether or not the player is actioning the event or if the vehicle has flipped, causing the ejection.

Return value: This represents whether or not they're allowed to leave the vehicle. 1 allows it, 0 blocks it.

19. `OnDamageLookup(receiving_obj, causing_obj, tagdata, tagname)` – This is called when damage needs to be done to an object. `receiving_obj` is always valid although `causing_obj` isn't and should be checked against -1 for validity. `tagdata` is the data for the damage to be applied and `tagname` is the type of damage being applied.

20. `OnWeaponAssignment(player, object, count, tag)` – This is called when a player is being assigned a weapon as they spawn. `count` refers to the weapon being assigned and is 0 for primary, 1 for secondary etc.

Return value: This represents the map id of the weapon to assign, return 0 if you don't wish to change it.

21. OnObjectCreation(m_objectId, player_owner, tag) – This is called when an object is being created. player_owner is the player who owns the object and is not always valid, check for validity before using.
22. OnClientUpdate(player, m_objectId) – This is called when a client sends the server an update packet. This includes things such as movement vectors, camera positions, button presses etc. This function is called frequently so care should be taken to keep processing to a minimum.

Note, while this isn't really an event notification it is a required function.

- GetRequiredVersion() – Phasor calls this function as the script loads (before OnScriptLoad) and it should return the version of Phasor the script is compatible with. This function being called does not guarantee the script will load. Phasor will reject it if the versions are incompatible.

Specific Things to Note:

Executing Server Commands:

You can execute server commands via scripts. This is done via the svcmd function. Lua doesn't really have data types and as such using this command is quite simple.

Let's say you want to kick a player. As Phasor doesn't explicitly have a kicking function you'd want to use svcmd.

```
svcmd("sv_kick " .. rconPlayer).
```

The ".." is used to build the string, it merges "sv_kick " with the data in rconPlayer. So, if we did:

```
local rconPlayer = 5  
svcmd("sv_kick " .. rconPlayer)
```

We'd effectively be doing: "sv_kick 5"

You need to take particular care when using svcmd. Phasor generally uses memory ids, not player ids. This causes an issue because svcmd operates on player ids. To solve this issue you need to call "resolveplayer", which will resolve a memory id into a player id. Below is an example of how to kick the player with memory id 5.

```
local rconPlayer = resolveplayer(5)  
svcmd("sv_kick " .. rconPlayer)
```

I've made another .pdf file regarding halo concepts and structures, it explains the difference between a memory and a player id (among other things).

<http://haloapps.wordpress.com>

That's all I really plan to explain in this guide. Hopefully you've read my other two guides and are starting to understand this stuff. If you haven't and are confused about things like memory, ids, tags etc then you should definitely give them a read.

If you want to learn more about Lua I highly recommend you read the official Lua site, <http://www.lua.org/> google usually finds results that are on this site. So, both work I guess.

I hope this guide makes the scripting stuff easier to understand. I released Phasor without much documentation and expected people to figure it out for themselves. That was stupid, but hopefully it's been fixed.

If you need additional help feel free to message me. It's probably easiest if you message me on the Phasor bug forum, which is at <http://phasor.proboards.com>

Thanks for reading and good luck...