Spring Lua Scripting Guide

How to write your own Gadgets, Widgets, Map Scripts, Unit Scripts and AI for Spring using Lua interfaces

Rough Draft - 23 Jan 2010

Table of Contents

INTRODUCTION

About this guide

The purpose of this guide is to pull together all of the sparse documentation and examples on Lua coding in Spring into a single comprehensive and easy-to-read reference. It provides basic theory as well as examples, tips, how-tos and reference sections to serve beginners and experts alike.

What it covers

All uses of Lua in the Spring game engine, namely: Lua Gadgets (Mods and player helpers), Lua Widgets (User Interface extensions), Map scripts and Lua AI.

What it does not cover

One important thing this manual won't cover is the Lua language itself. Lua is fairly easy to learn but this guide is **not** the place to do it. For that I recommend "Programming in Lua" which is a free book online. If you have experience with other languages you may be able to fudge your way through with minimal understanding of Lua but I wouldn't recommend it.

It doesn't cover operating system principles. If you don't know how to move files around, use a text editor or install new software then I'm not going to teach you. Even advanced stuff like using SVN is left to you to learn on your own.

This manual does not deal much with the internal details of the C->Lua interface bindings and not at all with C. For this I suggest you dig into the source code or find another guide (if one exists). This guide is only concerned with how the exposed interfaces are used by Lua scripts.

This is not an end-users guide to playing or cheating at Spring. It only covers topics of value to developers of Spring maps, mods, gadgets, widgets and AIs.

License

GLOSSARY

Spring uses a bit of jargon when talking about Lua scripts. Before you begin scripting it's very helpful if you understand the following concepts:

***A:** A catch-all term for mods based on original TA content.

BA: The mod Balanced Annihilation.

CA: The mod Complete Annihilation.

desync: The game is split into simulation code that runs on every computer and UI/helper code that only runs (or is different) for one player (see synced/unsynced). Even though a game can be hosted it is still basically peer-to-peer, in that every player has a local copy of the simulation state. A desync occurs when one or more players have different simulation data to everyone else. This results in players seeing a different outcome of events so the game would rapidly become chaos. In some mods desyncs can be caused by players not having the otacontent pack, but they can also be caused by bad engine or gadget code or differences in CPU behaviour between players.

call-in: A function that the gadget handler calls in each gadget that defines it. The gadget can perform some action before returning a response (if any) to Spring. A call-in in 'synced code' runs on the next 'sync frame' or as soon as possible if the code is unsynced.

call-out: A function that a gadget calls in the engine. The engine can perform some action before returning a response (if any) to the gadget.

c / c++ / cpp: The main programming language used by the Spring engine. It is very fast and powerful but hard to learn and modify 'on-the-fly'. Lua interacts with the C parts of the engine through defined interfaces. This means that Lua code cannot change something in the game engine

unless the engine provides a C->Lua interface for it. Fortuneately interfaces exist for most tasks a mod or script would conceivably want to do. If one doesn't exist you could request it (or add it yourself) but be aware that sometimes an interface wont be possible or even a good idea.

cob: Cob is custom language and subsystem used to program unit animations. A lua interface called LuaCob exists to manipulate this though it is now being obsoleted by unit scripts.

gadget: Lua script that provides new or changed functionality to a mod. In a way it's a "mod mod". A gadget can have synced and unsynced parts. Gadgets are similar to widgets except that widgets tend to be UI oriented while gadgets can change the simultation. Because they change the sim every player must have the gadget. Because of this most gadgets are bundled with mods or the engine. Gadgets require the mod have a luarules folder and a copy of the file gadget.lua, which can be found in any major mod.

lobby: Pre-game application used to find and create games. The most common are currently TASClient and SpringLobby.

Lua: Lua is a programming language used as a scripting language in Spring and a large number of other games and applications. It's wide use in games is due to its speed and low memory footprint. It is also very easy to embed and use. Spring uses Lua for all of its scriptable parts.

LuaGaia: Gaia refers to an interface available to lua scripts included with a map (eg, to spawn units, set custom victory conditions or change map physics).

LuaRules: The framework code that implements gadgets. It is required for gadgets to work. Most mods have a luarules folder and a file called gadgets.lua.

LuaUI: The framework that implements widgets. All code is unsynced.

mod: Spring is not actually a game, it is a game engine that can be extended using mods. So when you 'play Spring' what you are **actually** doing is playing a Spring mod (like Balanced Annihilation, Complete Annihilation, XTA, LLTA, etc..). The distinction is very important since a gadget or AI that works in one mod may not necessarily work in another.

widget: Lua script that runs only on the machine it is activated. Widgets can use only unsynced code and data such as functions that alter the UI in some way or issue orders to units. For these reasons it isn't necessary for all players to have a widget in order to use it.

sim: Short for simulation. In Spring jargon this is the data that represents the current state of the game for all players. It runs on every machine (including spectators) and must be constantly updated in a way that all players see the same results at the same time. When a game lags or desyncs this generally indicates that a player has fallen 'out of sync' or behind other players sim state and must 'catch up'. The only way it can generally do this is to slow everyone else down. Unstanding the sim is important if your mod is going to interact with it since any data you change or code you run on one computer must be synchronised across the network with all other players.

slowupdate: The engine performs some time-consuming calculations on each slowupdate. Slowupdates occur approximately every 15 frames (1/2 second) of game time.

Spring: Spring is the engine that runs spring mods (like Balanced Annihilation). It can not be played without mods (and vice versa). Spring is also the name of a module that is usually automatically available to scripts. It provides useful engine functions and constants.

synced code: Refers to when a gadget or widget runs the same code on the machines of every player. Syncing is required when changing part of the simulation (like a units properties) so all players have the same local data. If you don't the game desyncs.

sync frame: Represents a single update of the game sim. Each frame takes as long as is necessary for every machine in the game to synchronise all changes from the previous frame. In other words a sync frame is only as fast as the slowest computer or network in the game. Sync frames should not be confused with video frames-per-second as video FPS generally have no effect on other players.

TA: Short for "Total Annihilation", the game that inspired Spring and many of its mods.

unit script: A new lua interface to create unit animations.

unsynced code: Code that only runs on the local machine. Unsynced changes are not propagated to other clients so by design it cannot directly alter the simulation data. Unsynced code can generally only see or do something a player would be able to do (like issue orders to own units). A gadget or widget that runs only unsynced code does not need to be installed by every player so it is useful for helper AIs and UI changes.

CONVENTIONS

To make things clearer and to save repeating myself I have followed some basic conventions throughout this document.

Filenames and Paths

All paths and filenames are shown in *italic* unless they are in a code block. I always use the unix convention for path seperators which is a forward slash (/). If I am talking about a folder I will usually add a trailing slash even if it isn't strictly required. The path *YourMod/* always refers to the base directory of your mod and *Spring/* always refers to the directory where spring is installed.

Code

All code is shown in a monospaced font. Long snippets are placed between seperators like so:

```
------ -- -
Code example
------ -- -
```

When refering to code typed at the chat prompt (press Enter ingame) I add > to the front, eg:

```
> /cheat
> /devlua
> /luarules reload
```

TUTORIALS

This section is for those wanting to skip the theory and jump straight in, or get a handle on whats involved, or remind themselves how something is done. The tutorials focus on the how, not the why, so if you get confused please read the appropriate section of this manual for more details.

Creating a basic widget

This tutorial deals with creating a skeleton widget that does practically nothing.

Open SciTe, Notepad+ or your favorite text editor and create *[Spring]/LuaUI/Widgets/test.lua*. Enter the following:

```
------ -- -
function widget:GetInfo()
  return {
    name      = "Hello Widget",
    desc      = "Simple Widget Test",
    author    = "You",
    date      = "Jan 1, 2008",
    license   = "GNU GPL, v2 or later",
    layer     = 0,
    enabled   = false
  }
end

-- We define one local variable and one local function.
-- These cannot be accessed from outside this file but are fast to access.
-- The 'Spring' module was automatically included for us.

local hello = "Hello Spring"
local Echo = Spring.Echo

-- Now we create two call-ins.
-- The first will execute when the widget is loaded,
-- the second executes each time a unit is built.

function widget:Initialize()
  Echo( hello )
end

function widget:UnitCreated(unitID, unitDefID, unitTeam)
  Echo( "Hello Unit " .. unitID )
end
------ -- -
```

Now save and run Spring. Start a single-player game against an AI and Press <F11>. Turn on "Hello Widget" by clicking it then build something.

Congratulations on your first Spring widget!

GENERAL

This section deals with Lua usage that is consistent across gadgets, widgets and maps.

Local variables

Often you'll see things like

```
------ -- -
local GetUnitDefID = Spring.GetUnitDefID
------ -- -
```

This is purely a performance thing: by "localizing" a function or variable like this, Lua can access the function faster later on. You should also use local to define your own functions and variables when they are only being accessed further down in the same file or function.

INPUT

Keyboard

There are two methods for accessing keyboard input (not including commands typed in the chat prompt). You can setup a **KeyPress** and/or **KeyRelease** callin or you can query directly using the functions **Spring.GetKeyState** , **Spring.GetModKeyState** or **Spring.GetPressedKeys**. The method you use depends largely on how frequently you wish to check the input and how the logic fits with your existing code. The following two code segments are basically equivalent (check for ctrl + left mouse click):

```
------ -- -
-- You must import the KEYSYM table if you want to access keys by name
include('keysym.h.lua')

function widget:KeyPress(key, mods, isRepeat, label, unicode)
      if key == KEYSYMS.LCTRL or key == KEYSYMS.RCTRL then
            local dx, dy, leftPressed, middlePressed, rightPressed =
Spring.GetMouseState()
            if leftPressed then
                  MyCtrlClickHandler()
            end
      end
end
------ -- -

------ -- -
function widget:MousePress(x, y, button)
      local leftButton = 1
      if button == leftButton then
            local altPressed, ctrlPressed, metaPressed, shiftPressed =
Spring.GetModKeyState( )
            if ctrlPressed then
                  MyCtrlClickHandler()
            end
      end
end
------ -- -
```

Mouse

As with key events, mouse events can be caught with a callin or polled via a direct function call. The relevant callins are **MousePress, MouseRelease and MouseMove.** The functions are **Spring.GetMouseState** and **Spring.GetMouseStartPosition**. Check the reference section for details or the keyboard section above for example code.

FILESYSTEM

Virtual File System (VFS)

VFS Overview

Although Spring can access the filesystem directly (via os module) it is more common that you would want to access files included with your mod or Spring. Trouble is, most of these files are compressed into archives (.sdz/.sd7) so random access would generally be a difficult proceedure. Fortunately, the Spring Lua system automatically provides access to mod and base files via the VFS

module.

The VFS module doesn't simply open archives though. What it does is map your mod files, mod dependencies and Spring content onto a virtual file tree. All archives start from the 'roots' of the tree and share the same virtual space, meaning that if two or more archives contain the same resource file name the resources overlap and only one of the files will be retreived. Overlapping directories on the other hand are merged so the resulting virtual directory contains the contents of both. Here is an example of how this works:

Archive 1 (mods/mymod.sd7)	Archive 2 (base/otacontent.sdz)	VFS

textures
  |__ texture1.png
models
  |__ model1.mdl
	textures
	  |__ texture1.png
	  |__ texture2.png
	  |__ texture3.png
		textures
		  |__ texture1.png
		  |__ texture2.png
		  |__ texture3.png
		models
		  |__ model1.mdl

This raises the question: If both archives have a *texture1.png* then which *texture1.png* is retreived via the VFS? The answer depends on the order the archives are loaded and the **VFS mode** (more on modes in a minute). Generally however, each archive loaded overrides any archives loaded _before_ it. The standard order of loading (from first to last) is:

1.) The main *Spring/* game directory.
2.) The automatic dependencies *springcontent.sdz* and *maphelper.sdz*.
3.) Dependencies listed in your modinfo.lua (or modinfo.tdf), in the order listed.
4.) Your mod archive.

Loading lua files with VFS.Include()

```
VFS.Include('LuaUI/includes/filename.lua', [env], [vfsmode])
```

This loads and compiles the lua code from a file in the VFS.
The path is relative to the main Spring directory.
Env can be a table or nil. This is used as the starting environment. If nil then the env will be _G (global environment)
The vfsmode parameter defines the order in which archives are searched (see VFS Modes below)

Using VFS.Include() with a custom environment

```
env = { table = table, string = string, ...}
VFS.Include('filename.lua',env)
```

If the optional env argument is provided any non-local variables and functions defined in *filename.lua* are then accessable via env or _G. Vise-versa, any variables defined in env prior to passing to VFS.Include are available to code in the included file. Code running in *filename.lua* will see the contents of env in place of the normal _G environment.

Using the include() utility function (widgets only)

```
include('filename.lua')
```

The include() function is a thin wrapper around VFS.Include. It changes the search root to *LuaUI/* or *LuaUI/Headers/* for filenames ending in '.h.lua'.

VFS Modes

As stated earlier, the "stacking" behaviour of VFS is configurable at each call to a VFS retreival or listing function via a **mode** argument. All modes and their effects are listed in the reference section of this guide however as a rule a mode either restricts or prioritises files from certains types of archives (for example, VFS.RAW_FIRST will prioritise uncompressed files over compressed ones regardless of the order the archives were loaded.

Widget Config Data

Spring widgets have two callins that you can use to read/write a lua table to a shared config file. The configs are stored in *[spring]/LuaUI/Config/[modshortname].lua* which means widget configuration data is stored per-mod. If the names of the callins seem backwards to you try to remember that it is the widget handler getting or setting the configuration from your widget. Therefore **GetConfigData** is actually called to update the config file and **SetConfigData** is for setting values in the running widget.

GetConfigData

This optional callin is called when the widget handler wants to unload your widget and save its data. The return value is the data the handler (and therefore the config file) will get. In other words, the return value is saved to the config file.

```
------ -- -
local my_var = 'default value'

function widget:GetConfigData()
  return {
    my_saved_var = my_var
  }
end
------ -- -
```

SetConfigData

This is called when the widget handler loads or reloads your widget. The data argument passed in is the table that was stored in the config file (or an empty table).

```
------ -- -
local my_var = 'default value'

function widget:SetConfigData(data)
  my_var = data.my_saved_var or my_var
end
------ -- -
```

SOUNDS

Spring supports 3D sound, meaning you can 'place' a sound at a map position that gets louder when the camera is near it. You can also play global sounds that always play at the same volume regardless of the camera position.

PlaySoundFile() path is *Spring/Sounds* if you provide a filename with no path (eg. 'Sound.wav'), otherwise it is relative to *YourMod/* (eg. '*Sounds/Custom/Sound.wav*')

UNITS

Team vs. Allied Units

Don't get these confused. Your 'team' is all of the units directly under your control or part of an army when you are "comm sharing" (more than one player controlling the same army). Allied is when you are in a team with other players but each controls their own units (the units are different colors).

Unit IDs

Every individual unit in the game is given a unique integer ID. Many call-ins and unit functions use these IDs.

Unit Definitions (UnitDefs)

The engine automatically provides a read-only table called **UnitDefs** which is used to find a range of properties for each unit **type**. UnitDefs isn't exactly a normal table because it uses a metatable for access to its properties. Looping over the properties requires using the pairs class method like so:

```
------ -- -
for id,unitDef in pairs(UnitDefs) do
  for name,param in unitDef:pairs() do
    Spring.Echo(name,param)
  end
end
------ -- -
```

If you have a unitID and you want to get its unitdef use the following:

```
------ -- -
local defID = Spring.GetUnitDefID(unitID)
local def = UnitDefs[ defID ]

-- same as above in 1 line:
local def = UnitDefs[ Spring.GetUnitDefID(unitID) ]
------ -- -
```

Very useful is UnitDefs[i]["customParams"], since you can access any custom .FBI (unit file) param with it and use it in your Lua scripts. The descriptions of unit params can be found in the Reference Section of this manual.

Unit Animation / Scripting

About

Unit scripts replace the obsolete COB animation files. Unit scripts contain call-ins that are run when certain events happen to a unit. Events include the unit aiming, exploding, loading/unloading, etc. The primary purpose of catching these events is to perform animations, however you are not entirely limited to that.

Wiki Page: http://springrts.com/wiki/Animation-LuaScripting
Forum Thread: http://springrts.com/phpbb/viewtopic.php?f=12&t=20047
Example Scripts:

http://spring1944.svn.sourceforge.net/viewvc/spring1944/branches/MemberFolders/Tobi/S44LuaUnitScript.sdd/scripts/

Setup

Lua unit animation is implemented partially in the engine, and partially in a Lua gadget which is shipped with Spring in springcontent.sdz. By default, this gadget is not loaded, so a bit of setup is required to enable Lua unit scripts.

To enable Lua unit scripts, create a file *LuaRules/Gadgets/unit_script.lua*, and paste the following into it:

```
------ -- -
-- Enables Lua unit scripts by including the gadget from springcontent.sdz

-- Uncomment to override the directory which is scanned for *.lua unit scripts.
--UNITSCRIPT_DIR = "scripts/"

return VFS.Include("LuaGadgets/Gadgets/unit_script.lua")
------ -- -
```

Also, do check whether your *LuaRules/system.lua* is up to date, if you copied this verbatim into your game. (Must be newer then from 5 september 2009)

Now this is done, you can start putting *.lua files in your *scripts/* folder.

Unit Categories

TODO
what is the difference between "TEDClass" and "Category"?

Unit Commands (Orders)

TODO
Forum Thread: http://springrts.com/phpbb/viewtopic.php?f=23&t=12020

WEAPONS

Weapon Definitions (WeaponDefs)

The properties of all weapons are passed to Lua in the table 'WeaponDefs'. As with unitdefs each weapondef has a metatable so your must use the pairs() method to access them. The params are calculated from information stored in the mod. BA uses the file weapons.tdf and CA

```
------ -- -
for id,weaponDef in pairs(WeaponDefs) do
  for name,param in weaponDef:pairs() do
    Spring.Echo(name,param)
  end
end
```

```
------ -- -
```
to get each weapondef for a unitdef:
```
------ -- -
-- loop over all weapons on unit
for _, weapon in ipairs(unitDef.weapons) do
      local weaponDefID = weapon['weaponDef']
      local weaponDef = WeaponDefs[weaponDefID]
end
------ -- -
```

Damage and Armor

Weapons can deal different levels of damage depending on the target. The possible target types (or 'armor' types) are defined in the reference section. The damages dealt by each weapon are set in the TDF file using keys like 'VTOL' but in the WeaponDef they are an indexed array. To map one to the other requires code similar to this:

```
------ -- -
local sub_cats = {'l1subs','l2subs','l3subs','tl','atl'}
-- loop over target types
for _, target_cat in pairs(sub_cats) do
      -- damage values are arrays so we first need the index
      local cat_index = Game.armorTypes[target_cat]
      local damage = weaponDef.damages[cat_index] or 0
      if damage > 0 then
            table.insert(damage_list, damage)
      end
end
------ -- -
```

LIBRARIES

Using built-in libraries

When writing Lua code in Spring you can call on some libraries that ship with the engine. The most used is the '**Spring**' library which is automatically available from any script and provides many of the functions for interacting with the engine. Using it is simple, ie:

```
------ -- -
Spring.Echo('Spring library is always imported')
------ -- -
```

Another automatic import is the CMD table which provides numeric codes for most Spring unit commands. You also get a widgetHandler or gadgetHandler class and access to a set of libraries for virtual file operations, OpenGL drawing and more. The complete list of functions and libraries available to each script type can be found in the reference section of the manual.

In addition to Spring libraries you also have most of the libraries that ship with Lua such as **math** (extended with bit operations), **table, string** and limited versions of **os** and **package**.

Custom libraries written in Lua

Loading lua files with VFS.Include() or include()

```
VFS.Include('LuaUI/includes/filename.lua', [env], [vfsmode])
```

This topic is covered in detail in the VFS Filesystem section of this manual.

Custom libraries in other languages (advanced)

The Spring Lua interface retains Lua's **package** library. Through **package.loadlib** you can access DLLs (Windows) or shared libraries on Linux and OSX. The details of how to do this are in the regular Lua documentation. For the widest usage of your widget it is best to avoid platform-dependent languages like .Net / C#, Cocoa, etc and choose libraries, languages and dependencies that are cross-platform. For gadgets you don't really have a choice since it is essential that all players have the library. Use libraries in synced code with *extreme* caution because any subtle differences across platforms will desync or crash the game in a creeping fashion that makes debugging nearly impossible.

Note that any external library is bound to the widget or gadget rather than the engine. For this reason the library cannot directly perform any action on the engine internals. In general you'll just be passing data to you custom library and using the return value in spring functions.

GADGETS (LuaRules)

Gadgets are plug-ins added to mods to add new gamplay mechanics and options. Typically all players must have a gadget before it can be used. For this reason they are usually pre-packaged with mods so all players will have them. Gadgets are split into synced and unsynced parts (explained below). Because gadgets are generally part of the gameplay they cannot be toggled by players (though the game host may be able to disable it using mod options in the lobby).

Info

At the top of each gadget there is "function gadget:GetInfo()". This basically tells Spring some basic information about your gadget: what your gadget is called, who wrote it, etc. The easist way to do this is to copy it from an existing gadget and change the info accordingly.

Synced versus unsynced code

Your gadget runs in two modes: synced and unsynced. The simple way of putting it is that synced deals with things that affect *all* players (e.g., status of units), while unsynced deals with things that only affect a single player (e.g., GUI). gadgetHandler:IsSyncedCode() tells you which mode your gadget is currently in. You can do more when synced, but everyone has to run the code, not just the local player.

Many gadgets operate only in synced mode. This is why you often see things like

```
------ -- -
if (not gadgetHandler:IsSyncedCode()) then
  return false
end
------ -- -
```

Synced / Unsynced Protection

The LuaHandleSynced derivatives all use the same mechanism for sync protection.
1. The global namespace is synced
2. The synced code can not read from the unsynced code, but may send messages to it.
3. The unsynced code can read from the synced code using the SYNCED proxy table.

That table does not allow access to functions. There snext(), spairs(), and sipairs() functions can be

used on the SYNCED tables and its sub-tables.

Transfering variables between synced and unsynced code

Since synced and unsynced code cannot (and should not) see each others functions and variables Spring provides you with indirect methods to do so.

From synced to unsynced

SendToUnsynced(...)
RecvFromSynced(...)
So, say, you have my_synced_var set in some synced code, you call:
```
------ -- -
SendToUnsynced("someStringDescribingMyVar", my_synced_var)
------ -- -
```
then in unsynced code you have:
```
------ -- -
function RecvFromSynced(...)
  if arg[2] == "someStringDescribingMyVar" then
    local my_unsynced_var = arg[3]
  end
end
------ -- -
```
So you will now have the same value in my_unsynced_var as you did in my_synced_var

Caveats: Since the synced code is running on everyones computers don't send any data this way that shouldn't be seen by all players. It wouldn't be that hard for a cheating widget to intercept and use this data in unintended ways. By making something unsynced from synced you are effectively making it public. Keep in mind that if you want data that can only be accessed by synced code then you should think carefully about WHY it's only available there. 9 times out of 10 it will be because it would allow cheats.

From unsynced to synced

In the unsynced, there is SendLuaRulesMsg to SEND the message. Note that unlike the synced to unsynced process the message must be a string only. If you want to send variables you'll have to make them part of the string and decode them when they are received.

This function works by asking the server to rebroadcast your request to ALL clients. It is not just running on the local computer. You should avoid send excessive data or you risk creating network lag.
```
------ -- -
Spring.SendLuaRulesMsg("funkify unit:"..unitID)
------ -- -
```

In the synced part of a gadget, there is available a RecvLuaMsg call-in to RECEIVE the message. Call-ins are described elsewhere in the manual but the basic code is:
```
------ -- -
function gadget:RecvLuaMsg(msg, playerID)
  -- Message could be anything, we need to see if it's the right one
  if string.find(msg,"funkify unit:") then
    -- yep, this is our message, get the unitID by removing the rest
    unitID = msg:gsub("funkify unit:","")
    Spring.Echo("Unit "..unitID.. " got da funk!")
  end
end
------ -- -
```

Caveats: The same call-in runs on all clients and receives all messages from all players so it's up to you to sort out the messages using string parsing and the player id.

Call-ins

This is the meat of a gadget. These tell your gadget when something happens, and allow your gadget to take action (and sometimes return a result). A callin looks something like this:

```
------ -- -
function gadget:CallinName(...)
  --your code here
end
------ -- -
```

Whenever an event corresponding to the callin happens, the callin gets called. Some callins are called with arguments giving information about the event. For example, when gadget:GameFrame(n) is called, n is the number of the frame. Furthermore, some callins expect that you return some value. For example, callins with "Allow" in their name typically block the action if you return a false value.
Callins are listed in gadgets.lua and in the reference section of this guide.

MAP SCRIPTS (LuaGaia)

Map scripts are similar to Gadgets except they are included with a map rather than a mod.

[TODO: Learn about LuaGaia]

WIDGETS (LuaUI)

Widgets are plug-ins which typically add user interface features or special unit behaviours. The primary differences from gadgets are:
* Widgets are installed and activated seperately for each player.
* Widgets always run unsynced.
* Widgets can receive direct user input
* Widgets can directly perform opengl drawing operations
* Widgets have filtered or restricted access to data on unseen enemy units

Info

At the top of each widget there is "function widget:GetInfo()". This basically tells Spring's widget handler some basic information about your widget: what your widget is called, who wrote it, etc. The easist way to do this is to copy it from an existing widget and change the info accordingly. The following table explains each field:

Widget Info Options

Key	Required	Default	Description
name	Yes		Name displayed in widget list
desc	?		Describes widget in tooltips
author	?		Name of developers(s)
date	?		Date last updated (freeform text)
license	?		Type of license (freeform text)

layer	?	0	Integer which determines the order widgets are loaded, called and displayed. Higher numbers will run later and draw GUI elements above lower numbers.
enabled	?	False	Whether the widget is on by default.

Installing widgets

Place the widget lua file in *Spring/LuaUI/Widgets/*. Make sure if you downloaded the widget it is unzipped and any additional files it needs are placed according the the widget authors instructions.

Activating and deactivating widgets

All widgets are enabled and disabled by pressing [F11] in game. Widgets are then toggled by clicking on the widget name. Green widgets are active, Red are inactive, and Orange means the widget tried to activate but failed due to an error.

You can also use:

```
> /luaui togglewidget  <widgetname>
> /luaui enablewidget  <widgetname>
> /luaui disablewidget <widgetname>
```

DEBUGGING

Logging

All errors, print statements, and Spring.Echo log to **Spring/infolog.txt**. The numbers in square brackets at the start of most lines is the frame number.

Stop on errors

Some errors are easier to catch if you stop processing. The error function (from standard Lua) can be used to trigger your own exceptions. **error** is superior to **Spring.Echo** for catching errors because it will also report the file and line number where the error occured.

Debug commands

```
> /cheat
```

Enables you to use commands to spawn units and remove fog and other things generally considered to be cheats. [TODO, list commands in reference]

```
> /devlua
```

Allows you to perform interesting testing actions like editing Defs. Use with care as some actions will cause desynces in multiplayer. [TODO, list commands in reference]

Reloading scripts

To reload a gadget without restarting Spring use :

```
> /luarules reload
```

To reload your widgets use :

```
> /luaui reload
```

Though I found this crashed a lot. A safer way is use F11 and click on a widget to toggle or:

```
> /luaui togglewidget <widgetname>
> /luaui togglewidget <widgetname>
```

If you bind that to a key or command it makes testing much easy.

Bigger console

IceUI and the Message Seperator widgets both allow resizeable console and fonts which will let you see more of your debug output and even scroll back through it. Highly recommended.

Advanced debugging

Using Tracebacks

Lua has a function debug.traceback() that allows you to see all of the function calls that led up to the current one. This is useful information to send to the log when you aren't sure why a function was called or where bad data may have come from.

```
------ -- -
-- Print a traceback if the arguments are nil
local function DistVec(v1,v2)
      if (v1 == nil or v2 == nil) then
            Spring.Echo("ERROR: nil values passed: " .. debug.traceback())
      end
      ...
end
------ -- -
```

Example Trackback:

```
[string "D:\Games\Spring\LuaUI\Widgets\unit_scout.lu..."]:124: in function
'DistVec'
[string "D:\Games\Spring\LuaUI\Widgets\unit_scout.lu..."]:299: in function
<[string "D:\Games\Spring\LuaUI\Widgets\unit_scout.lu..."]:298>
[C]: in function 'sort'
[string "D:\Games\Spring\LuaUI\Widgets\unit_scout.lu..."]:303: in function
'GetAreasByDistanceFrom'
[string "D:\Games\Spring\LuaUI\Widgets\unit_scout.lu..."]:310: in function
'FindBestArea'
[string "D:\Games\Spring\LuaUI\Widgets\unit_scout.lu..."]:322: in function
'PatrolBestArea'
[string "D:\Games\Spring\LuaUI\Widgets\unit_scout.lu..."]:387: in function
<[string "D:\Games\Spring\LuaUI\Widgets\unit_scout.lu..."]:381>
[C]: in function 'pcall'
[string "LuaUI/widgets.lua"]:592: in function 'UnitIdle'
[string "LuaUI/widgets.lua"]:1620: in function <[string
"LuaUI/widgets.lua"]:1618>
(tail call): ?
```

More Information

This post has some information on debugging scripts: http://spring.clan-sy.com/phpbb/viewtopic.php?f=23&t=13932

My util_globals include file [TODO: release somewhere] provides a dump function to pretty-print nested tables.

PERFORMANCE

jK has done some interesting Lua performance tests. The results can be found at CA's LuaPerformance Wiki . I have reproduced a couple of tips below.

Localise variables

If a global variable (upvalue) is going to be used regularly (like in a loop) it usually pays to localise it. It is common to do this with functions in the Spring module. eg:

```
------ -- -
local spEcho = Spring.Echo
------ -- -
```

Use built-in methods

Built-in string methods are faster than the string library. Therefore:

```
------ -- -
string.find(stringVar, "%.abc")
------ -- -
```

should be:

```
------ -- -
stringVar:find("%.abc")
------ -- -
```

Use numeric loops where possible

Looping on pairs or ipairs has extra overhead compared to numeric loops

```
------ -- -
for i,v in pairs(array) do.. end
------ -- -
```

should be:

```
------ -- -
for i=1,#array do local v = array[i]; .. end
------ -- -
```

Provide default values using or

Using if can be expensive however it is possible to define alternate values inline

```
------ -- -
if foo == nil then foo = 'bar' end
------ -- -
```

should be:

```
------ -- -
foo = foo or 'bar'
------ -- -
```

REFERENCE

This section provides function definitions and data tables for the Lua interface.

Lua Class Tree

```
LuaHandle
|
|- LuaHandleSynced
| |
| |- LuaGaia (map)
| '- LuaRules (gadget)
|
`- LuaUI (widget)
```

Access Modes Table

Access modes restrict access to call-ins that would otherwise allow players to cheat. The table below defines the player groups allowed to run the call-back for each combination of script type and access restriction. Players groups also include Gaia (the map) when the script is run from LuaGaia.

TODO: Find out whether these modes actually do anything yet or are planned for a future version.

	userMode	readFull	readAllyTeam	ctrlFull	ctrlTeam	selectTeam
Gadget	false	true	ALL	true	ALL	ALL
Map	false	false	Gaia	false	Gaia	Gaia
Widget	true	false	PlayerTeam	false	PlayerTeam	PlayerTeam
Widget (spectator)	true	true	ALL	false	NONE	depends*

Function Library Access Table

Functions are divided into libraries and each library has access restrictions shown in this table.

Library / Interface	LuaUI	LuaRules (unsynced)	LuaRules (synced)	Gaia (unsynced)	Gaia (synced)
Lua_ConstGame	+	+	+		
Lua_UnitDefs	+	+	+		
Lua_WeaponDefs	+	+	+		
Lua_FeatureDefs	+	+	+		
Lua_CMDs	+	+	+		
Lua_UnsyncedRead	+	+	-		
Lua_UnsyncedCtrl	+	+	+		
Lua_SyncedRead	+**	+	+		
Lua_SyncedCtrl	-	-	+		
Lua_MoveCtrl	-	-	+		
Lua_PathFinder	+	+	+		
Lua_OpenGL_Api	+	+	-		

Lua_GLSL_Api	+	+	-		
Lua_FBO_and_RBO	+	+	-		
Lua_UnitRendering	-	+	-		
Lua_ConstGL	+	+	-		
Lua_VFS	+	+	+*		
Lua_Scream	+	+	-		
Lua_BitOps	+	+	+		

* only VFS.ZIP_ONLY
** with special LOS handling and decoy unit handling

Debugging Functions

Some useful debugging commands:

General Debug Commands

`print( msg )`
Output message to stdout and log

`Spring.Echo( msg )`
Output message to screen and log

Widget Debug Commands

`PrintCommandQueue(uid)`
Output a units command queue to stdout

`PrintGroups()`
Output your units to stdout, organised into groups

`PrintTeamUnits(team)`
Output your units to stdout, organised into groups

Unit States

The game does not appear to define Lua contants for unit states. Below the state is given as the key name from Spring.GetUnitStates() and also the CMD constant name for Spring.GiveOrderToUnit(). The number is the actual value used when querying or setting the state.

UnitState["movestate"] | CMD.MOVE_STATE
0 = Hold Position
1 = Manuever
2 = Roam

UnitState["firestate"] | CMD.FIRE_STATE
0 = Hold Fire
1 = Return Fire
2 = Fire At Will
3 = Shoot Neutrals!

UnitState["repeat"] | CMD.REPEAT

0 = Off
1 = On

UnitState["cloak"] | CMD.CLOAK
0 = Off
1 = On

UnitState["active"] | CMD.ONOFF
0 = Off
1 = On

UnitState["trajectory"] | CMD.TRAJECTORY
0 = Off
1 = On

UnitState["autoland"]
0 = Off
1 = On

UnitState["autorepairlevel"] | CMD.AUTOREPAIRLEVEL

UnitState["loopbackattack"] | CMD.LOOPBACKATTACK

Call-in Access Quick Reference

Not all call-ins are available from all code. The table below shows where a call-in is valid.

Call-in	LuaUI	LuaRules (unsynced)	LuaRules (synced)	Gaia (unsynced)	Gaia (synced)
CommandNotify	+				
Update	+				
Initialize	+				
UnitIdle	+				

* only VFS.ZIP_ONLY

Widget Handler Actions List

```
widget:LoadOrderList()
widget:SaveOrderList()
widget:LoadConfigData()
widget:SaveConfigData()
widget:SendConfigData()
widget:Initialize()
widget:LoadWidget(filename, fromZip)
widget:NewWidget()
widget:FinalizeWidget(widget, filename, basename)
widget:ValidateWidget(widget)
widget:InsertWidget(widget)
widget:RemoveWidget(widget)
widget:UpdateCallIn(name)
widget:UpdateWidgetCallIn(name, w)
widget:RemoveWidgetCallIn(name, w)
widget:UpdateCallIns()
widget:EnableWidget(name)
widget:DisableWidget(name)
widget:ToggleWidget(name)
widget:RaiseWidget(widget)
widget:LowerWidget(widget)
widget:FindWidget(name)
widget:RegisterGlobal(owner, name, value)
widget:DeregisterGlobal(owner, name)
widget:SetGlobal(owner, name, value)
widget:RemoveWidgetGlobals(owner)
widget:GetHourTimer()
widget:GetViewSizes()
widget:ForceLayout()
widget:ConfigLayoutHandler(data)
```

Widget Call-in List

```
widget:Shutdown()
widget:Update()
widget:ConfigureLayout(command)
widget:CommandNotify(id, params, options)
widget:AddConsoleLine(msg, priority)
widget:GroupChanged(groupID)
widget:CommandsChanged()
widget:SetViewSize(vsx, vsy)
widget:ViewResize(vsx, vsy)
widget:DrawScreen()
widget:DrawGenesis()
widget:DrawWorld()
widget:DrawWorldPreUnit()
widget:DrawWorldShadow()
widget:DrawWorldReflection()
widget:DrawWorldRefraction()
widget:DrawScreenEffects(vsx, vsy)
widget:DrawInMiniMap(xSize, ySize)
widget:KeyPress(key, mods, isRepeat, label, unicode)
widget:KeyRelease(key, mods, label, unicode)
widget:WidgetAt(x, y)
widget:MousePress(x, y, button)
```

```
widget:MouseMove(x, y, dx, dy, button)
widget:MouseRelease(x, y, button)
widget:MouseWheel(up, value)
widget:IsAbove(x, y)
widget:GetTooltip(x, y)
widget:GamePreload()
widget:GameStart()
widget:GameOver()
widget:TeamDied(teamID)
widget:TeamChanged(teamID)
widget:PlayerChanged(playerID)
widget:GameFrame(frameNum)
widget:ShockFront(power, dx, dy, dz)
widget:WorldTooltip(ttType, ...)
widget:MapDrawCmd(playerID, cmdType, px, py, pz, ...)
success, newReady = widget:GameSetup(state, ready, playerStates)
result = widget:DefaultCommand(...)
widget:UnitCreated(unitID, unitDefID, unitTeam)
widget:UnitFinished(unitID, unitDefID, unitTeam)
widget:UnitFromFactory(unitID, unitDefID, unitTeam, factID, factDefID,
userOrders)
widget:UnitDestroyed(unitID, unitDefID, unitTeam)
widget:UnitTaken(unitID, unitDefID, unitTeam, newTeam)
widget:UnitGiven(unitID, unitDefID, unitTeam, oldTeam)
widget:UnitIdle(unitID, unitDefID, unitTeam)
widget:UnitCommand(unitID, unitDefID, unitTeam, cmdId, cmdOpts, cmdParams)
widget:UnitCmdDone(unitID, unitDefID, unitTeam, cmdID, cmdTag)
widget:UnitDamaged(unitID, unitDefID, unitTeam, damage, paralyzer)
widget:UnitEnteredRadar(unitID, unitTeam)
widget:UnitEnteredLos(unitID, unitTeam)
widget:UnitLeftRadar(unitID, unitTeam)
widget:UnitLeftLos(unitID, unitTeam)
widget:UnitEnteredWater(unitID, unitDefID, unitTeam)
widget:UnitEnteredAir(unitID, unitDefID, unitTeam)
widget:UnitLeftWater(unitID, unitDefID, unitTeam)
widget:UnitLeftAir(unitID, unitDefID, unitTeam)
widget:UnitSeismicPing(x, y, z, strength)
widget:UnitLoaded(unitID, unitDefID, unitTeam, transportID, transportTeam)
widget:UnitUnloaded(unitID, unitDefID, unitTeam, transportID, transportTeam)
widget:UnitCloaked(unitID, unitDefID, unitTeam)
widget:UnitDecloaked(unitID, unitDefID, unitTeam)
widget:UnitMoveFailed(unitID, unitDefID, unitTeam)
bool = widget:RecvLuaMsg(msg, playerID)
widget:StockpileChanged(unitID, unitDefID, unitTeam, weaponNum, oldCount,
newCount)
```

Gadget Handler Actions List

```
gadget:EnableGadget(name)
gadget:DisableGadget(name)
gadget:ToggleGadget(name)
gadget:RaiseGadget(gadget)
gadget:LowerGadget(gadget)
gadget:FindGadget(name)
gadget:RegisterGlobal(owner, name, value)
gadget:DeregisterGlobal(owner, name)
gadget:SetGlobal(owner, name, value)
gadget:RemoveGadgetGlobals(owner)
gadget:GetHourTimer()
gadget:GetViewSizes()
gadget:RegisterCMDID(gadget, id)
```

Gadget Call-in List

```
gadget:GamePreload()
gadget:GameStart()
gadget:Shutdown()
gadget:GameFrame(frameNum)
gadget:RecvFromSynced(...)
gadget:GotChatMsg(msg, player)
gadget:RecvLuaMsg(msg, player)
gadget:SetViewSize(vsx, vsy)
gadget:ViewResize(vsx, vsy)
gadget:GameOver()
gadget:TeamDied(teamID)
gadget:DrawUnit(unitID, drawMode)
gadget:AICallIn(dataStr)
gadget:CommandFallback(unitID, unitDefID, unitTeam,
gadget:AllowCommand(unitID, unitDefID, unitTeam,
gadget:AllowUnitCreation(unitDefID, builderID,
gadget:AllowUnitTransfer(unitID, unitDefID,
gadget:AllowUnitBuildStep(builderID, builderTeam,
gadget:AllowFeatureCreation(featureDefID, teamID, x, y, z)
gadget:AllowResourceLevel(teamID, res, level)
gadget:AllowResourceTransfer(teamID, res, level)
gadget:AllowDirectUnitControl(unitID, unitDefID, unitTeam,
gadget:MoveCtrlNotify(unitID, unitDefID, unitTeam, data)
gadget:TerraformComplete(unitID, unitDefID, unitTeam,
gadget:UnitCreated(unitID, unitDefID, unitTeam, builderID)
gadget:UnitFinished(unitID, unitDefID, unitTeam)
gadget:UnitFromFactory(unitID, unitDefID, unitTeam,
gadget:UnitDestroyed(unitID,     unitDefID,     unitTeam,
gadget:UnitExperience(unitID, unitDefID, unitTeam,
gadget:UnitIdle(unitID, unitDefID, unitTeam)
gadget:UnitCmdDone(unitID, unitDefID, unitTeam, cmdID, cmdTag)
gadget:UnitDamaged(unitID, unitDefID, unitTeam,
gadget:UnitTaken(unitID, unitDefID, unitTeam, newTeam)
gadget:UnitGiven(unitID, unitDefID, unitTeam, oldTeam)
gadget:UnitEnteredRadar(unitID, unitTeam, allyTeam, unitDefID)
gadget:UnitEnteredLos(unitID, unitTeam, allyTeam, unitDefID)
gadget:UnitLeftRadar(unitID, unitTeam, allyTeam, unitDefID)
gadget:UnitLeftLos(unitID, unitTeam, allyTeam, unitDefID)
gadget:UnitSeismicPing(x, y, z, strength,
gadget:UnitLoaded(unitID, unitDefID, unitTeam,
gadget:UnitUnloaded(unitID, unitDefID, unitTeam,
gadget:UnitCloaked(unitID, unitDefID, unitTeam)
gadget:UnitDecloaked(unitID, unitDefID, unitTeam)
gadget:StockpileChanged(unitID, unitDefID, unitTeam,
gadget:FeatureCreated(featureID, allyTeam)
gadget:FeatureDestroyed(featureID, allyTeam)
gadget:ProjectileCreated(proID, proOwnerID)
gadget:ProjectileDestroyed(proID)
gadget:Explosion(weaponID, px, py, pz, ownerID)
gadget:Update()
gadget:DefaultCommand(type, id)
gadget:DrawGenesis()
gadget:DrawWorld()
gadget:DrawWorldPreUnit()
gadget:DrawWorldShadow()
gadget:DrawWorldReflection()
gadget:DrawWorldRefraction()
gadget:DrawScreenEffects(vsx, vsy)
gadget:DrawScreen(vsx, vsy)
gadget:DrawInMiniMap(mmsx, mmsy)
gadget:KeyPress(key, mods, isRepeat, label, unicode)
gadget:KeyRelease(key, mods, label, unicode)
```

```
gadget:MousePress(x, y, button)
gadget:MouseMove(x, y, dx, dy, button)
gadget:MouseRelease(x, y, button)
gadget:MouseWheel(up, value)
gadget:IsAbove(x, y)
gadget:GetTooltip(x, y)
```

Call-in Functions

Call-ins are registered in gadgets, widgets and maps to catch game and UI events. Not every call-in is available in each mode and some are only valid for synced or unsynced use. To use a call-in simply define it in your script as a non-local function and the widget or gadget handler will see this and register the call-in with the underlying lua interface.

NOTE: When using the functions listed here replace **handler** with **widget** or **gadget** depending on where you use it. Check the tables above for details of which contexts a callin supports.

AddConsoleLine

```
function gadget/widget:AddConsoleLine(text)
```

When text is entered into the console (like with Spring.Echo) this callback occurs.

AllowCommand

```
function gadget/widget:AllowCommand(unitID, unitDef, team, command, parameters, options)
```

AllowCommand is called when the command is given, before the unit's queue is altered. The return value is whether it should be let into the queue (I think it blocks it if any gadget returns false and doesn't ask others after that). The queue remains untouched when a command is blocked, whether it would be queued or replace the queue.
AllowCommand intercepts all commands. Use it to check whether targets are valid, etc.
AllowCommand happens on the exact moment a command is given.
Example:

```
------ -- -
function gadget/widget:AllowCommand(unitID, unitDef, team, command, parameters, options)
   if command == CMD_PEANUT_BUTTER then
      if memberJellyGroup(unitID) then
         return true
      end
   end
return false
end
------ -- -
```

AllowUnitCreation

AllowUnitTransfer

AllowUnitBuildStep

AllowFeatureCreation

AllowFeatureBuildStep

AllowResourceLevel

AllowResourceTransfer

CobCallback

CommandFallback

```
function gadget/widget:CommandFallBack(unitID, unitDef, team, command,
parameters, options)
```

CommandFallback is called when the unit reaches an unknown command in its queue (i.e. one not handled by the engine). It returns the two values used and finished, if no gadget returns used as true the command is dropped since it seems noone knows it, if a gadget returns true,true the command is removed because it's done, with true,false it's kept in the queue and CommandFallback gets called again on the next slowupdate.

Example:

```
------ -- -
function gadget/widget:CommandFallBack(unitID, unitDef, team, command,
parameters, options)
   if command == CMD_PEANUT_BUTTER then
      DoSomethingNiftyHere()
      return true, true;
   end
return false
end
------ -- -
```

CommandNotify

ConfigureLayout

DrawGenesis

DrawWorld

DrawWorldPreUnit

DrawWorldShadow

DrawWorldReflection

DrawWorldRefraction

DrawScreenEffects

DrawScreen

DrawInMiniMap

Explosion

FeatureCreated

FeatureDestroyed

GameFrame

```
function gadget/widget:GameFrame( frameNumber )
```

Called every sync frame (if synced) or as fast as possible (unsynced). The frame number could be used as a crude timer or to only run on frame one, or to do something every n'th frame.

GameLoadLua

Called at the end of the loading process, so heavy computing doesn't need to be done at gamestart.

GameOver

Called when the the game is won or lost.

GameStart

Called when the game is about to start. After GameLoadLua().

GetTooltip

GroupChanged

Initialize

```
function gadget/widget:Initialize()
```
This is called before the game proper starts, specifically when "LuaRules" shows on the loading screen.

IsAbove

KeyPress

```
function gadget/widget:KeyPress(key, mods, isRepeat, label, unicode)
```
key: The KEYSYMS code (see keysyms reference).
mods: Modifier keys being pressed (mods.alt, mods.ctrl, mods.meta, mods.shift). Each is a boolean state.
isRepeat: Is true on the second and subsequent calls if the key is being held down.

Called repeatedly when a key is pressed down. If you want an action to occur only once check for isRepeat == false.

KeyRelease

LayoutButtons

MouseMove

MousePress

MouseRelease

PlayerChanged

```
function gadget/widget:PlayerChanged(playerID)
```
Called when a player becomes a spectator. This is a good place to disable your widget if it does nothing in spectator mode.

```
------ -- -
function widget:PlayerChanged(playerID)
      -- Disable if switched to spectator
      if GetSpectatingState() then
            widgetHandler:RemoveWidget()
            return
      end
end
------ -- -
```

PlayerRemoved

```
function gadget/widget:PlayerRemoved(playerID)
```
Presumably when a player leaves the game.

ShockFront

Shutdown

```
function gadget/widget:Shutdown()
```
Called when the widget or gadget is turned off.

TeamChanged

```
function gadget/widget:TeamChanged(teamID)
```
Under some cicumstances (such as with cheats on) it is possible for a player to change teams. This is called when that happens.

TeamDied

```
function gadget/widget:TeamDied(teamID)
```
Called when a team is wiped out.

UnitCreated

UnitDestroyed

```
function gadget/widget:UnitDestroyed(unitID, unitDefID, unitTeam, attackerID, attackerDefID, attackerTeam)
```

UnitFinished

UnitFromFactory

UnitTaken

UnitGiven

UnitIdle

UnitCommand

UnitSeismicPing

UnitEnteredRadar

UnitEnteredLos

UnitLeftRadar

UnitLeftLos

UnitLoaded

UnitUnloaded

UnitEnteredWater

UnitEnteredAir

UnitLeftWater

UnitLeftAir

Update

function gadget/widget:Update()

Runs as often as possible. This can be useful for regular polling or timed events. The example below shows how to spread updates out over a set amount of time. This can save considerable CPU cycles if your updates are intensive.

You should make sure you don't poll for an event via Update when a more specific call-in could be used, since most of the time call-ins do their polling (if any) in C++ and are therefore faster.

```
------ -- -
local updatePeriod = 0.5 -- half a second
local lastUpdate = 0

function widget:Update()

   -- skip updates if they are too close together
   local now = GetGameSeconds()
   if (now < lastUpdate + updatePeriod) then
```

```
      return -- skip the update
   end
   lastUpdate = now

   -- do stuff ...
end
------ -- -
```

WorldTooltip

Unit Script Call-ins

Introduction

Call-ins are calls from the engine, into the unit script. In other words, these functions are called, if they are defined, when a particular “event” happens. For Lua unit scripts, a new callin mechanism has been implemented, which is faster than the regular callin mechanism which is used for widgets and gadgets.

Types for arguments are only shown where they're ambiguous. For a number of (common) arguments, the types are:

* unitID: number
* piece: number
* axis: number (1 = x axis, 2 = y axis, 3 = z axis)
* heading/pitch: number (radians)

Generic

```
Create ( ) -> nil
```
This is called just after the unit script has been created.

```
StartMoving ( ) -> nil
StopMoving ( ) -> nil
```
These are called whenever the unit starts or stops moving. Typical use for them is to trigger wheels to start spinning, animate treads, or start/stop a walking animation.

```
Killed ( number recentDamage, number maxHealth ) -> number corpseType
```
This is called when the unit is killed. The severity of the kill may be calculated as severity=recentDamage/maxHealth. Typically, this function would play a death animation for the unit, and finally return a corpse type.

```
WindChanged ( number heading, number strength ) -> nil
```
This is called for wind generators whenever the wind direction or strength changes.

```
ExtractionRateChanged ( number rate ) -> nil
```
Called for metal extractors each time their extraction rate (metal per second) changes.

```
setSFXoccupy ( number curTerrainType ) -> nil
```
Called when terrain type changes. Terrain type is calculated with the following rules (in this order):

* If unit is being transported -> curTerrainType = 0
* If ground height < -5 and unit is always upright -> curTerrainType = 2
* If ground height < -5 and unit is not always upright -> curTerrainType = 1
* If ground height < 0 and unit is always upright -> curTerrainType = 1

* Otherwise -> curTerrainType = 4

Where is curTerrainType = 3 ? :-)
Candidate to be changed to something saner later on.

```
MoveRate ( number curRate ) -> nil
```
Called only for certain types of aircraft (those which use CTAAirMoveType.) The move rate is determined by the following rules (in this order):

* If the aircraft is landing or taking off -> curRate = 1
* Otherwise -> curRate = floor(curSpeed / maxSpeed * 3), clamped to be in the range [0, 2]

```
QueryLandingPads ( ) -> { number piece1, number piece2, ... }
```
Called one time for units with landing pads. Should return a table with pieces which should be used as landing pads. The number of pieces returned also determines the number of pads, so for Lua unit scripts there is no QueryLandingPadCount.

```
Activate ( ) -> nil
Deactivate ( ) -> nil
```
Exact situation these are called depends a lot on the unit type. Factories are activated when they open their yard and deactivated when they close it. Aircraft are activated when they take off and deactivated when the land. Any unit that can be turned on or off is activated when it's turned on and deactivated when it's turned off. On SetUnitValue(COB.ACTIVATION, ...) one of these call-ins may be called too.

Weapons

Weapon functions come in two variants.

* Separate function with numeric weapon number suffix. (e.g. AimWeapon1(heading, pitch))
* Combined function which gets weapon number as second argument. (e.g. AimWeapon(weaponNum, heading, pitch))

Only the first variant is listed here, so whenever you see a function whose name ends with a numeric one ("1"), you should either replace it with the actual weapon number, or you can write a single combined function that takes a weaponNum argument. For each script, all functions should use the same variant. Either all using name suffix, or all using combined functions.

```
QueryWeapon1 ( ) -> number piece
AimFromWeapon1 ( ) -> number piece
AimWeapon1 ( heading, pitch ) -> boolean
AimShield1 ( ) -> boolean
```

Weapon support. The return value of QueryWeapon and AimFromWeapon determines the pieces which will be used for aiming: typically e.g. the barrel for QueryWeapon and the turret for AimFromWeapon. AimWeapon is then called to allow the script to turn the weapons in the target direction, which is passed as the heading and pitch argument (in radians). Only if AimWeapon returns true, the weapon is actually fired.

TODO: better explanation (?), note about AimWeapon being called very often etc.

```
FireWeapon1 ( ) -> nil
Shot1 ( ) -> nil
EndBurst1 ( ) -> nil
```

If after aiming the unit actually fires it's weapon, FireWeapon is called once at the beginning of the salvo. Shot is called just before each projectile is fired. RockUnit (see below) is called just after all projectiles for that frame have been fired. At the end of the salvo, EndBurst is called.

Of these call-ins, FireWeapon is the more generic one and Shot and EndBurst are more specialized. FireWeapon is usually used to play a recoil animation or emit some smoke near the weapon's barrel using EmitSfx.

```
BlockShot1 ( targetUnitID, boolean userTarget ) -> boolean
```
Allows you to block a weapon from shooting. TargetUnitID may be nil: in this case the unit has a ground-attack order.

```
TargetWeight1 ( targetUnitID ) -> number
```
Allows you to tweak the priority of the target for this particular weapon. The target priority is multiplied by the return value of this function. Lower priority targets are targeted first, so return a value smaller than 1 to prioritize a target, or return a value larger than 1 to avoid a target.

The exact behavior of specific values shouldn't be relied upon.

```
RockUnit ( x, z ) -> nil
```
A bit like the weapon-specific FireWeapon function, although this is called after any weapon fires. As argument it gets a two dimensional vector in the direction the unit just fired. This may be used to "rock" the unit as a whole a bit as part of the firing animation. Note though that this vector is in world space, so for a truly realistic rock direction it needs to be rotated according to the unit's current heading.

```
HitByWeapon ( x, z, weaponDefID, damage ) -> nil | number newDamage
```
This is called if a unit has been hit by a weapon. (x, z) is the direction from which the projectile came in unit space (the reverse direction of the impulse, to be exact.) It also gets the weaponDefID of the weapon which is dealing the damage, and the amount of damage. If HitByWeapon returns a number, this number will replace the damage value calculated by the engine.

Note: these call-in runs just before the LuaRules UnitPreDamaged callin (see also LuaCallinReturn). If HitByWeapon overrides the damage, UnitPreDamaged will see the new damage value, and may override again the damage value.

Builders and factories

```
StartBuilding ( heading, pitch ) -> nil
StartBuilding ( ) -> nil
StopBuilding ( ) -> nil
```
These notify the script when a builder or factory starts or stops building.

The first variant (with heading and pitch arguments) is called for builders. For factories, the second variant is used. In this case the heading and pitch are not necessary, because the factory script specifies the build location itself using QueryBuildInfo.

```
QueryBuildInfo ( ) -> number piece
```
For factories only. Should return the piece to which the unit which is going to be build will be attached while it's being build.

```
QueryNanopiece ( ) -> number piece
```
Called each time a nano particle is to be created. Should return the piece at which the particle will be spawned. This may be used to iterate through a few pieces, to simulate the factory/builder having

multiple nanolathes.

Transports

There are some different code paths inside Spring related to transports, each (unfortunately) also associated with a different set of unit script call-ins.

This table shows for the three different transportUnloadMethods for both air transports and ground transports which callins are used and roughly when and how often they are called. Refer to the documentation below for a description of the call-ins.

The entire behavior around transports should be considered subject to change; it is obvious it is far from a perfect design currently.

	Load	UnloadLand (0)	UnloadDrop (1)	UnloadLandFlood (2)
		default unload	fly over and drop unit	land, then release all units at once
air	BeginTransport (each) QueryTransport (each)	EndTransport (last)	EndTransport (each)	StartUnload (first) TransportDrop (each) EndTransport (last)
ground	TransportPickup (each)	TransportDrop (each)	TransportDrop (each)	TransportDrop (each) EndTransport (last)

Air transports

```
BeginTransport ( passengerID ) -> nil
QueryTransport ( passengerID ) -> number piece
```

For an air transport, if any one unit is picked up, these two are called in succession and the passenger is attached to the piece returned by the second one.

```
StartUnload ( ) -> nil
```

Only called in UnloadLandFlood behavior. Signals the start of an unload cycle.

```
TransportDrop ( passengerID, x, y, z ) -> nil
```

Only called in UnloadLandFlood behavior. Called when a passenger will be unloaded. Contrary to ground transports, Spring will detach the passenger after the call.

```
EndTransport ( ) -> nil
```

In UnloadLand and UnloadLandFlood behaviors, these are called one time after all units are unloaded. (The transport is empty.) For the UnloadDrop behavior, this is called for every unit that is unloaded.

Ground transports

```
TransportPickup ( passengerID ) -> nil
```

Called when a passenger should be loaded into the transport. This should eventually call AttachUnit to actually attach the unit. Assuming the transport is in range of the next passenger, this will be called again for the next passenger 16 frames later, unless the script enters the BUSY state: then

Spring will not move on to the next passenger until the script leaves the BUSY state.

```
TransportDrop ( passengerID, x, y, z ) -> nil
```
Called when a passenger should be unloaded. This should eventually call DetachUnit to actually detach the unit, unless the used unload method is UnloadLandFlood, in which case Spring will actually detach the unit after the call.

```
EndTransport ( ) -> nil
```
Only called in UnloadLandFlood behavior, after the last unit has been unloaded.

Passenger

```
Falling ( ) -> nil
```
For a unit dropped from an UnloadDrop transport, this is called every frame to inform the script the unit is still falling. It may be used to show a parachute for example.

```
Landed ( ) -> nil
```
This is called one time after the unit reached the ground. May be used to hide a parachute for example.

Internal

These call-ins are **NOT** available to Lua unit scripts. They are called by the engine however, but always 'intercepted' by the framework gadget. For completeness (or if you are poking at the gadget), they are listed here anyway.

```
MoveFinished ( piece, axis ) -> nil
TurnFinished ( piece, axis ) -> nil
```

Called when a move or turn finished. The framework gadget uses this to resume the coroutines which are waiting for the particular move or turn to be finished.

```
Destroy (  ) -> nil
```
Called right before the unit's script is destroyed. This may happen if the unit is being destroyed, but also if the unit's script is being replaced by another script. The framework gadget uses this to stop all threads and remove the unit from some internal data structures.

Unit Script Call-outs

Introduction

The new callouts are defined in the Spring.UnitScript table.

Types for arguments are only shown where they're ambiguous. For a number of (common) arguments, the types are:

* unitID: number
* piece: number
* axis: number (1 = x axis, 2 = y axis, 3 = z axis)
* destination: number (world coors or radians)
* speed: number (world coors or radians per second)
* accel/decel: number (radians per second per frame)

Animation

```
Spring.UnitScript.SetPieceVisibility ( piece, boolean visible ) -> nil
Spring.UnitScript.Show ( piece ) -> nil
Spring.UnitScript.Hide ( piece ) -> nil
```
These may be used to show/hide pieces of the unit's model.

```
Spring.UnitScript.Move ( piece, axis, destination[, speed] ) -> nil
```
Move piece along axis to the destination position. If speed is given, the piece isn't moved immediately, but will move there at the desired speed. The X axis is mirrored compared to BOS/COB scripts, to match the direction of the X axis in Spring world space.

```
Spring.UnitScript.Turn ( piece, axis, destination[, speed] ) -> nil
```
Turn piece around axis to the destination angle. If speed is given, the piece isn't rotated immediately, but will turn at the desired angular velocity. Angles are in radians.

```
Spring.UnitScript.Spin ( piece, axis, speed[, accel] ) -> nil
```
Makes piece spin around axis at the desired angular velocity. If accel is given, the piece does not start at this velocity at once, but will accelerate to it. Both negative and positive angular velocities are supported. Accel should always be positive, even if speed is negative.

```
Spring.UnitScript.StopSpin ( piece, axis[, decel] ) -> nil
```
Stops a piece from spinning around the given axis. If decel is given, the piece does not stop at once, but will decelerate to it. Decel should always be positive. This function is similar to Spin(piece, axis, 0, decel), however, StopSpin also frees up the animation record.

```
Spring.UnitScript.IsMoving ( piece, axis ) -> boolean
Spring.UnitScript.IsTurning ( piece, axis ) -> boolean
Spring.UnitScript.IsSpinning ( piece, axis ) -> boolean
```
Returns true if the piece is moving along the axis, or turning/spinning around the axis.

```
Spring.UnitScript.GetPieceTranslation ( piece ) -> number x, y, z
Spring.UnitScript.GetPieceRotation ( piece ) -> number x, y, z
```
Get the current translation/rotation of a piece. The returned numbers match the values passed into Move and Turn.

```
Spring.UnitScript.GetPiecePosDir ( piece ) -> number px, py, pz, dx, dy, dz
```
Get the piece's position (px, py, pz) and direction (dx, dy, dz) in unit space. This is quite similar to Spring.GetUnitPiecePosDir, however that function returns in world space.

Threads

```
Spring.UnitScript.StartThread ( function fun, ... ) -> nil
```
Starts a new (animation) thread, which will execute the function 'fun'. All arguments except the function to run are passed as-is as arguments to 'fun'. COB-Threads has a decent description on COB threads, which are mimicked here in Lua using coroutines.

```
Spring.UnitScript.SetSignalMask ( number mask ) -> nil
Spring.UnitScript.Signal ( number signal ) -> nil
```
These two support functions offer a powerful way to kill running threads. SetSignalMask assigns a mask to the currently running thread (any new threads started by this one will inherit the signal mask). Signal immediately stops all threads of this unit for which the bitwise and of mask and signal is not zero.

```
Spring.UnitScript.WaitForMove ( piece, axis ) -> nil
```

```
Spring.UnitScript.WaitForTurn ( piece, axis ) -> nil
```
Waits until the piece has stopped moving along / turning around the axis. If the piece is not animating, this functions return at once. You can not wait for a spin, because a spin does never finish.

```
Spring.UnitScript.Sleep ( number milliseconds ) -> nil
```
Waits a number of milliseconds before returning.

Effects

```
Spring.UnitScript.EmitSfx ( piece, number id ) -> nil
```
Emits a particle from the given piece. The id may be one of: TODO

```
Spring.UnitScript.Explode ( piece, number flags ) -> nil
```
Explodes a piece, optionally creating a particle which flies off. Typically used inside Killed. Explode does not hide the piece by itself; if using it outside Killed you may want to Hide the piece immediately after. The flags may be a combination of: TODO

Other

```
Spring.UnitScript.AttachUnit ( piece, passengerID ) -> nil
Spring.UnitScript.DetachUnit ( passengerID ) -> nil
```
Attaches or detaches another unit (a passenger, as this is designed for transports) to this unit. For AttachUnit, piece specifies the attachment point.

```
Spring.UnitScript.GetUnitValue ( ... ) -> number | number, number
Spring.UnitScript.SetUnitValue ( ... ) -> nil
```
This may be used instead of COB's get and set codes. It is recommended however, to use dedicated Lua functions (e.g. Spring.GetUnitHealth(unitID) instead of GetUnitValue(4)): these functions should be considered a relic of the past.

Note that these are identical to Spring.GetUnitCOBValue and Spring.SetUnitCOBValue (see also Lua_SyncedCtrl#Lua_to_COB for the signature of those functions), except that an unitID argument shouldn't be passed to the versions in the Spring.UnitScript table.

```
Spring.UnitScript.GetLongestReloadTime ( unitID ) -> number reloadTime
```
Returns max(reload time) of all the unit's weapons in milliseconds. This utility function exists to aid in porting BOS scripts, which have a SetMaxReloadTime call-in that is called immediately after Create.

Internal

You generally shouldn't need any of the following call-outs, they are provided by the engine to allow the Lua framework to do it's work.

```
Spring.UnitScript.CreateScript ( unitID, table callIns ) -> nil
```
This deletes the unit's current script (whether it's a COB or Lua script doesn't matter), and sets it up with a brand new Lua unit script, initially registering the call-ins given in the table as (string, function) pairs.

```
Spring.UnitScript.UpdateCallIn ( unitID, string name[, function callIn] ) -> nil
```
This updates a single call-in for the unit's current Lua unit script. If the unit does not currently have a Lua unit script, an error is raised. If the callIn argument is not given (or nil), the call-in with the given name is removed.

```
Spring.UnitScript.CallAsUnit ( unitID, number team, function fun, ... ) -> nil
```
As none of the call-outs takes a unitID, the engine needs to know the active unit when one of those

is called. Using this function another function can be called with the active unit set arbitrarily.

`Spring.UnitScript.SetDeathScriptFinished ( [number wreckLevel] )`
Tells Spring the Killed script finished, and which wreckLevel to use. If wreckLevel is not given no wreck is created. May only be called after Killed has been called. DO NOT USE, the framework handles this transparently (it passes the return value of Killed into this function.)

Lua FeatureDefs

The FeatureDefs[] table holds all information about the features used in a mod. Note: Its entries are metatables, so you can't use the pairs() iterator on them, use this instead:

```
------ -- -
for id,featureDef in pairs(FeatureDefs) do
  for name,param in featureDef:pairs() do
    Spring.Echo(name,param)
  end
end
------ -- -
```

Here is an example of a FeatureDef:

```
------ -- -
FeatureDefs[3]["blocking"] = false,
FeatureDefs[3]["burnable"] = false,
FeatureDefs[3]["deathFeature"] = "",
FeatureDefs[3]["destructable"] = false,
FeatureDefs[3]["drawType"] = 0,
FeatureDefs[3]["drawTypeString"] = "3do",
FeatureDefs[3]["energy"] = 0,
FeatureDefs[3]["filename"] = "features/corpses/type1.tdf",
FeatureDefs[3]["floating"] = false,
FeatureDefs[3]["geoThermal"] = false,
FeatureDefs[3]["height"] = 3.9283447265625,
FeatureDefs[3]["hitSphereOffsetX"] = 0,
FeatureDefs[3]["hitSphereOffsetY"] = 0,
FeatureDefs[3]["hitSphereOffsetZ"] = 0,
FeatureDefs[3]["hitSphereScale"] = 1,
FeatureDefs[3]["id"] = 3,
FeatureDefs[3]["mass"] = 66.900001525879,
FeatureDefs[3]["maxHealth"] = 397,
FeatureDefs[3]["maxx"] = 24,
FeatureDefs[3]["maxy"] = 3.9283447265625,
FeatureDefs[3]["maxz"] = 24,
FeatureDefs[3]["metal"] = 68,
FeatureDefs[3]["midx"] = 0,
FeatureDefs[3]["midy"] = 1.9383087158203,
FeatureDefs[3]["midz"] = 0,
FeatureDefs[3]["minx"] = -24,
FeatureDefs[3]["miny"] = -0.051727294921875,
FeatureDefs[3]["minz"] = -24.318237304688,
FeatureDefs[3]["modelType"] = 0,
FeatureDefs[3]["modelname"] = "objects3d/3X3D",
FeatureDefs[3]["name"] = "ahermes_heap",
FeatureDefs[3]["noSelect"] = false,
FeatureDefs[3]["radius"] = 27.287155151367,
FeatureDefs[3]["reclaimable"] = true,
FeatureDefs[3]["reclaimTime"] = 23213,
FeatureDefs[3]["tooltip"] = "Wreckage",
FeatureDefs[3]["upright"] = false,
FeatureDefs[3]["useHitSphereOffset"] = false,
FeatureDefs[3]["xsize"] = 6,
```

```
FeatureDefs[3]["ysize"] = 6,
------ -- -
```

Lua WeaponDefs

The WeaponDefs[] table holds all information about the weapons used in a mod. Note: Its entries are metatables, so you can't use the pairs() iterator on them, use this instead:

```
------ -- -
for id,weaponDef in pairs(WeaponDefs) do
  for name,param in weaponDef:pairs() do
    Spring.Echo(name,param)
  end
end
------ -- -
```

Here an example of a weapon table:

```
------ -- -
WeaponDefs[3]["accuracy"] = 0,
WeaponDefs[3]["alwaysVisible"] = false,
WeaponDefs[3]["areaOfEffect"] = 128,
WeaponDefs[3]["avoidFriendly"] = false,
WeaponDefs[3]["beamburst"] = false,
WeaponDefs[3]["beamtime"] = 1,
WeaponDefs[3]["beamTTL"] = 1,
WeaponDefs[3]["beamDecay"] = 1,
WeaponDefs[3]["bouncerebound"] = 1,
WeaponDefs[3]["cameraShake"] = 1.5,
WeaponDefs[3]["canAttackGround"] = true,
WeaponDefs[3]["cegTag"] = "",
WeaponDefs[3]["collisionSize"] = 0.050000000745058,
WeaponDefs[3]["coreThickness"] = 0.25,
WeaponDefs[3]["coverageRange"] = 0,
WeaponDefs[3]["cylinderTargetting"] = 0,
WeaponDefs[3]["damages"] = {
     [1] = 420,
     [2] = 420,
     [3] = 420,
     [4] = 420,
     ...
     -> [ armorType ] = number damage,
     ["paralyzeDamageTime"] = 0,
     ["impulseBoost"] = 0.12300000339746,
     ["impulseFactor"] = 0.12300000339746,
     ["craterBoost"] = 0,
     ["craterMult"] = 0,
   }
WeaponDefs[3]["dance"] = 0,
WeaponDefs[3]["description"] = "CruiserDepthCharge",
WeaponDefs[3]["dropped"] = false,
WeaponDefs[3]["duration"] = 0.050000000745058,
WeaponDefs[3]["edgeEffectiveness"] = 0.80000001192093,
WeaponDefs[3]["energyCost"] = 0,
WeaponDefs[3]["explosionSpeed"] = 3.233583688736,
WeaponDefs[3]["exteriorShield"] = false,
WeaponDefs[3]["filename"] = "TorpedoLauncher",
WeaponDefs[3]["fireSound"] = {
     [1] = {
       ["id"] = 8,
       ["name"] = "torpedo1.wav",
       ["volume"] = 14.491376876831,
     }
   }
WeaponDefs[3]["fireStarter"] = 0,
WeaponDefs[3]["graphicsType"] = -16777216,
```

```
WeaponDefs[3]["gravityAffected"] = false,
WeaponDefs[3]["groundbounce"] = true,
WeaponDefs[3]["groundslip"] = 1,
WeaponDefs[3]["guided"] = true,
WeaponDefs[3]["hardStop"] = false,
WeaponDefs[3]["heightBoostFactor"] = -1,
WeaponDefs[3]["heightMod"] = 0.20000000298023,
WeaponDefs[3]["hitSound"] = {
     [1] = {
       ["id"] = 9,
       ["name"] = "xplodep2.wav",
       ["volume"] = 28.982753753662,
     }
   }
WeaponDefs[3]["id"] = 3,
WeaponDefs[3]["intensity"] = 0.89999997615814,
WeaponDefs[3]["interceptedByShieldType"] = 0,
WeaponDefs[3]["interceptor"] = 0,
WeaponDefs[3]["isShield"] = false,
WeaponDefs[3]["largeBeamLaser"] = false,
WeaponDefs[3]["laserFlareSize"] = 15,
WeaponDefs[3]["leadLimit"] = 1,
WeaponDefs[3]["leadBonus"] = 1,
WeaponDefs[3]["manualFire"] = false,
WeaponDefs[3]["maxAngle"] = 180,
WeaponDefs[3]["maxVelocity"] = 200,
WeaponDefs[3]["metalCost"] = 0,
WeaponDefs[3]["minIntensity"] = 0,
WeaponDefs[3]["movingAccuracy"] = 0,
WeaponDefs[3]["name"] = "advdepthcharge",
WeaponDefs[3]["noAutoTarget"] = false,
WeaponDefs[3]["noExplode"] = false,
WeaponDefs[3]["noFeatureCollide"] = false,
WeaponDefs[3]["noFriendlyCollide"] = true,
WeaponDefs[3]["noSelfDamage"] = true,
WeaponDefs[3]["numbounce"] = 1,
WeaponDefs[3]["onlyForward"] = true,
WeaponDefs[3]["onlyTargetCategories"] = {
     ["antiflame"] = false,
     ["vtol"] = false,
     ["notland"] = false,
     ["fort"] = false,
     ["special"] = false,
     ["notair"] = false,
     ["kbot"] = false,
     ["antiemg"] = false,
     ["commander"] = false,
     ["jam"] = false,
     ["tport"] = false,
     ["constr"] = false,
     ["strategic"] = false,
     ["kamikaze"] = false,
     ["minelayer"] = false,
     ["hover"] = false,
     ["noweapon"] = false,
     ["plant"] = false,
     ["ship"] = false,
     ["antilaser"] = false,
     ["phib"] = false,
     ["mine"] = false,
     ["notstructure"] = false,
     ["tank"] = false,
     ["mobile"] = false,
     ["underwater"] = false,
```

```
      ["antigator"] = false,
      ["notship"] = false,
      ["all"] = false,
      ["notsub"] = false,
      ["weapon"] = false,
    }
WeaponDefs[3]["paralyzer"] = false,
WeaponDefs[3]["predictBoost"] = 1,
WeaponDefs[3]["projectilespeed"] = 6.6666665077209,
WeaponDefs[3]["proximityPriority"] = 1.0,
WeaponDefs[3]["range"] = 500,
WeaponDefs[3]["reload"] = 6,
WeaponDefs[3]["restTime"] = 0,
WeaponDefs[3]["salvoDelay"] = 0.10000000149012,
WeaponDefs[3]["salvoSize"] = 1,
WeaponDefs[3]["selfExplode"] = true,
WeaponDefs[3]["selfprop"] = true,
WeaponDefs[3]["shieldAlpha"] = 0.20000000298023,
WeaponDefs[3]["shieldBadColorB"] = 0.5,
WeaponDefs[3]["shieldBadColorG"] = 0.5,
WeaponDefs[3]["shieldBadColorR"] = 1,
WeaponDefs[3]["shieldEnergyUse"] = 0,
WeaponDefs[3]["shieldForce"] = 0,
WeaponDefs[3]["shieldGoodColorB"] = 1,
WeaponDefs[3]["shieldGoodColorG"] = 0.5,
WeaponDefs[3]["shieldGoodColorR"] = 0.5,
WeaponDefs[3]["shieldInterceptType"] = 0,
WeaponDefs[3]["shieldMaxSpeed"] = 0,
WeaponDefs[3]["shieldPower"] = 0,
WeaponDefs[3]["shieldPowerRegen"] = 0,
WeaponDefs[3]["shieldPowerRegenEnergy"] = 0,
WeaponDefs[3]["shieldRadius"] = 0,
WeaponDefs[3]["shieldRepulser"] = false,
WeaponDefs[3]["size"] = 3.0499999523163,
WeaponDefs[3]["sizeGrowth"] = 0.20000000298023,
WeaponDefs[3]["smartShield"] = false,
WeaponDefs[3]["soundTrigger"] = false,
WeaponDefs[3]["sprayAngle"] = 0,
WeaponDefs[3]["startvelocity"] = 3.6666667461395,
WeaponDefs[3]["stockpile"] = false,
WeaponDefs[3]["supplyCost"] = 0,
WeaponDefs[3]["sweepFire"] = false,
WeaponDefs[3]["targetBorder"] = 0,
WeaponDefs[3]["targetMoveError"] = 0,
WeaponDefs[3]["targetable"] = 0,
WeaponDefs[3]["tdfId"] = 0,
WeaponDefs[3]["thickness"] = 2,
WeaponDefs[3]["tracks"] = true,
WeaponDefs[3]["trajectoryHeight"] = 0,
WeaponDefs[3]["turnRate"] = 0.031319729983807,
WeaponDefs[3]["turret"] = false,
WeaponDefs[3]["twoPhase"] = false,
WeaponDefs[3]["type"] = "TorpedoLauncher",
WeaponDefs[3]["uptime"] = 10,
WeaponDefs[3]["visibleShield"] = false,
WeaponDefs[3]["visibleShieldHitFrames"] = 0,
WeaponDefs[3]["visibleShieldRepulse"] = false,
WeaponDefs[3]["visuals"] = {
      ["colorR"] = 1,
      ["colorB"] = 0,
      ["colorG"] = 0,
      ["beamWeapon"] = false,
      ["sizeDecay"] = 0,
      ["tileLength"] = 200,
```

```
      ["smokeTrail"] = false,
      ["pulseSpeed"] = 1,
      ["renderType"] = 1,
      ["alphaDecay"] = 1,
      ["color2B"] = 1,
      ["separation"] = 1,
      ["scrollSpeed"] = 5,
      ["color2R"] = 1,
      ["modelName"] = "DEPTHCHARGE",
      ["noGap"] = 1,
      ["color2G"] = 1,
      ["stages"] = 5,
    }
WeaponDefs[3]["vlaunch"] = false,
WeaponDefs[3]["waterbounce"] = true,
WeaponDefs[3]["waterWeapon"] = true,
WeaponDefs[3]["weaponAcceleration"] = 0.016666667535901,
WeaponDefs[3]["wobble"] = 0,
------ -- -
```

Lua UnitDefs

The UnitDefs[] table holds all information about the units used in the mod. Note: Its entries are metatables, so you can't use the pairs() iterator on them, use this instead:

```
------ -- -
for id,unitDef in pairs(UnitDefs) do
  for name,param in unitDef:pairs() do
    Spring.Echo(name,param)
  end
end
------- -- -
```

Very useful is UnitDefs[i]["customParams"], since you can access any custom fbi param with it and use it in your lua scripts.
Because the UnitDefs tables are very large, I simply print here an example of it (BA5.8 core commander):

```
------ -- -
UnitDefs[216]["customParams"] = {
   } //Variable names written must be lowercase and variables can only be
strings.
UnitDefs[216]["TEDClass"] = "COMMANDER",
UnitDefs[216]["activateWhenBuilt"] = true,
UnitDefs[216]["aihint"] = 216,
UnitDefs[216]["cobID"] = -1,
UnitDefs[216]["airLosRadius"] = 21.09375,
UnitDefs[216]["airStrafe"] = true,
UnitDefs[216]["armorType"] = 8,
UnitDefs[216]["armoredMultiple"] = 1,
UnitDefs[216]["autoHeal"] = 2.6666667461395,
UnitDefs[216]["buildDistance"] = 128,
UnitDefs[216]["buildOptions"] = {
     [1] =UnitDefID1,
     [2] =UnitDefID2,
     ...
   }
UnitDefs[216]["buildRange3D"] = true,
UnitDefs[216]["buildSpeed"] = 300,
UnitDefs[216]["buildTime"] = 75000,
UnitDefs[216]["buildangle"] = 0,
UnitDefs[216]["builder"] = true,
UnitDefs[216]["buildingDecalDecaySpeed"] = 0.10000000149012,
UnitDefs[216]["buildingDecalSizeX"] = 4,
UnitDefs[216]["buildingDecalSizeY"] = 4,
UnitDefs[216]["buildingDecalType"] = 0,
```

```
UnitDefs[216]["buildpicname"] = "CORCOM.DDS",
UnitDefs[216]["canAssist"] = true,
UnitDefs[216]["canAttack"] = true,
UnitDefs[216]["canBeAssisted"] = true,
UnitDefs[216]["canBuild"] = true,
UnitDefs[216]["canCapture"] = true,
UnitDefs[216]["canCloak"] = true,
UnitDefs[216]["canDGun"] = true,
UnitDefs[216]["canDropFlare"] = false,
UnitDefs[216]["canFight"] = true,
UnitDefs[216]["canFly"] = false,
UnitDefs[216]["canGuard"] = true,
UnitDefs[216]["canHover"] = false,
UnitDefs[216]["canKamikaze"] = false,
UnitDefs[216]["canLoopbackAttack"] = false,
UnitDefs[216]["canMove"] = true,
UnitDefs[216]["canPatrol"] = true,
UnitDefs[216]["canReclaim"] = true,
UnitDefs[216]["canRepair"] = true,
UnitDefs[216]["canRepeat"] = true,
UnitDefs[216]["canRestore"] = true,
UnitDefs[216]["canResurrect"] = false,
UnitDefs[216]["canSelfRepair"] = false,
UnitDefs[216]["canSubmerge"] = false,
UnitDefs[216]["capturable"] = false,
UnitDefs[216]["captureSpeed"] = 300,
UnitDefs[216]["cloakCost"] = 100,
UnitDefs[216]["cloakCostMoving"] = 1000,
UnitDefs[216]["controlRadius"] = 32,
UnitDefs[216]["deathExplosion"] = "COMMANDER_BLAST",
UnitDefs[216]["decloakDistance"] = 50,
UnitDefs[216]["decloakOnFire"] = true,
UnitDefs[216]["decloakSpherical"] = true,
UnitDefs[216]["decoyDef"] = nil,
UnitDefs[216]["dlHoverFactor"] = -1,
UnitDefs[216]["drag"] = 0.13046109676361,
UnitDefs[216]["energyCost"] = 2500,
UnitDefs[216]["energyMake"] = 25,
UnitDefs[216]["energyStorage"] = 0,
UnitDefs[216]["energyUpkeep"] = 0,
UnitDefs[216]["extractRange"] = 0,
UnitDefs[216]["extractsMetal"] = 0,
UnitDefs[216]["factoryHeadingTakeoff"] = false,
UnitDefs[216]["fallSpeed"] = 1,
UnitDefs[216]["filename"] = "units/corcom.fbi",
UnitDefs[216]["flankingBonusMode"] = 0,
UnitDefs[216]["flankingBonusMax"] = 0,
UnitDefs[216]["flankingBonusMin"] = 0,
UnitDefs[216]["flankingBonusDirX"] = 0,
UnitDefs[216]["flankingBonusDirY"] = 0,
UnitDefs[216]["flankingBonusDirZ"] = 0,
UnitDefs[216]["flankingBonusMobilityAdd"] = 0,
UnitDefs[216]["flareDelay"] = 0.30000001192093,
UnitDefs[216]["flareDropVectorX"] = 0,
UnitDefs[216]["flareDropVectorY"] = 0,
UnitDefs[216]["flareDropVectorZ"] = 0,
UnitDefs[216]["flareEfficiency"] = 0.5,
UnitDefs[216]["flareReloadTime"] = 5,
UnitDefs[216]["flareSalvoDelay"] = 0,
UnitDefs[216]["flareSalvoSize"] = 4,
UnitDefs[216]["flareTime"] = 90,
UnitDefs[216]["floater"] = false,
UnitDefs[216]["frontToSpeed"] = 0.10000000149012,
UnitDefs[216]["fullHealthFactory"] = false,
```

```
UnitDefs[216]["gaia"] = "",
UnitDefs[216]["health"] = 3000,
UnitDefs[216]["height"] = 42.333557128906,
UnitDefs[216]["hideDamage"] = true,
UnitDefs[216]["highTrajectoryType"] = 0,
UnitDefs[216]["hitSphereOffsetX"] = 0,
UnitDefs[216]["hitSphereOffsetY"] = 0,
UnitDefs[216]["hitSphereOffsetZ"] = 0,
UnitDefs[216]["hitSphereScale"] = 1,
UnitDefs[216]["holdSteady"] = true,
UnitDefs[216]["hoverAttack"] = false,
UnitDefs[216]["humanName"] = "Commander",
UnitDefs[216]["iconType"] = "corcommander",
UnitDefs[216]["id"] = 216,
UnitDefs[216]["idleAutoHeal"] = 2.6666667461395,
UnitDefs[216]["idleTime"] = 1800,
UnitDefs[216]["isAirBase"] = false,
UnitDefs[216]["isBomber"] = false,
UnitDefs[216]["isBuilder"] = true,
UnitDefs[216]["isBuilding"] = false,
UnitDefs[216]["isCommander"] = true,
UnitDefs[216]["isFactory"] = false,
UnitDefs[216]["isFeature"] = false,
UnitDefs[216]["isFighter"] = false,
UnitDefs[216]["isFirePlatform"] = false,
UnitDefs[216]["isGroundUnit"] = false,
UnitDefs[216]["isMetalExtractor"] = false,
UnitDefs[216]["isMetalMaker"] = false,
UnitDefs[216]["isTransport"] = false,
UnitDefs[216]["jammerRadius"] = 0,
UnitDefs[216]["kamikazeDist"] = 0,
UnitDefs[216]["leaveTracks"] = false,
UnitDefs[216]["levelGround"] = true,
UnitDefs[216]["loadingRadius"] = 220,
UnitDefs[216]["losHeight"] = 20,
UnitDefs[216]["losRadius"] = 28.125,
UnitDefs[216]["makesMetal"] = 0,
UnitDefs[216]["mass"] = 5000,
UnitDefs[216]["maxAcc"] = 0.18000000715256,
UnitDefs[216]["maxAileron"] = 0.014999999664724,
UnitDefs[216]["maxBank"] = 0.80000001192093,
UnitDefs[216]["maxDec"] = 0.037500001490116,
UnitDefs[216]["maxElevator"] = 0.0099999997764826,
UnitDefs[216]["maxFuel"] = 0,
UnitDefs[216]["maxHeightDif"] = 14.558809280396,
UnitDefs[216]["maxPitch"] = 0.44999998807907,
UnitDefs[216]["maxRepairSpeed"] = 300,
UnitDefs[216]["maxRudder"] = 0.0040000001899898,
UnitDefs[216]["maxSlope"] = 0.93969261646271,
UnitDefs[216]["maxThisUnit"] = 5000,
UnitDefs[216]["maxWaterDepth"] = 35,
UnitDefs[216]["maxWeaponRange"] = 300,
UnitDefs[216]["maxx"] = 17.456253051758,
UnitDefs[216]["maxy"] = 42.333557128906,
UnitDefs[216]["maxz"] = 22.837493896484,
UnitDefs[216]["metalCost"] = 2500,
UnitDefs[216]["metalMake"] = 1.5,
UnitDefs[216]["metalStorage"] = 0,
UnitDefs[216]["metalUpkeep"] = 0,
UnitDefs[216]["midx"] = 0,
UnitDefs[216]["midy"] = 19.558204650879,
UnitDefs[216]["midz"] = 0,
UnitDefs[216]["minAirBasePower"] = 0,
UnitDefs[216]["minCollisionSpeed"] = 1,
```

```
UnitDefs[216]["minWaterDepth"] = -10000000,
UnitDefs[216]["minx"] = -17.456253051758,
UnitDefs[216]["miny"] = -3.2171478271484,
UnitDefs[216]["minz"] = -9.9750061035156,
UnitDefs[216]["modCategories"] = {
     ["notship"] = false,
     ["notair"] = false,
     ["core"] = false,
     ["commander"] = false,
     ["all"] = false,
     ["level10"] = false,
     ["ctrl_c"] = false,
     ["notsub"] = false,
     ["weapon"] = false,
   }
UnitDefs[216]["model"] = {
     ["textures"] = {
     }
     ["type"] = "3do",
     ["name"] = "CORCOM",
     ["path"] = "objects3d/CORCOM",
   }
UnitDefs[216]["moveData"] = {
     ["name"] = "KBOT3",
     ["type"] = "ground",
     ["maxSlope"] = 0.41221475601196,
     ["id"] = 2,
     ["depthMod"] = 0.10000000149012,
     ["crushStrength"] = 50,
     ["family"] = "kbot",
     ["depth"] = 5000,
     ["slopeMod"] = 9.680196762085,
     ["size"] = 4,
   }
UnitDefs[216]["moveType"] = 0,
UnitDefs[216]["myGravity"] = 0.40000000596046,
UnitDefs[216]["name"] = "corcom",
UnitDefs[216]["nanoColorB"] = 0.20000000298023,
UnitDefs[216]["nanoColorG"] = 0.69999998807907,
UnitDefs[216]["nanoColorR"] = 0.20000000298023,
UnitDefs[216]["needGeo"] = false,
UnitDefs[216]["noAutoFire"] = false,
UnitDefs[216]["noChaseCategories"] = {
     ["all"] = false,
   }
UnitDefs[216]["onOffable"] = false,
UnitDefs[216]["pieceTrailCEGTag"] = "",
UnitDefs[216]["pieceTrailCEGRange"] = 0,
UnitDefs[216]["power"] = 2541.6667480469,
UnitDefs[216]["radarRadius"] = 700,
UnitDefs[216]["radius"] = 24.740692138672,
UnitDefs[216]["reclaimSpeed"] = 300,
UnitDefs[216]["reclaimable"] = false,
UnitDefs[216]["refuelTime"] = 5,
UnitDefs[216]["releaseHeld"] = false,
UnitDefs[216]["repairSpeed"] = 300,
UnitDefs[216]["resurrectSpeed"] = 300,
UnitDefs[216]["seismicRadius"] = 0,
UnitDefs[216]["seismicSignature"] = 0,
UnitDefs[216]["selfDCountdown"] = 5,
UnitDefs[216]["selfDExplosion"] = "COMMANDER_BLAST",
UnitDefs[216]["shieldWeaponDef"] = nil,
UnitDefs[216]["showNanoFrame"] = true,
UnitDefs[216]["showNanoSpray"] = true,
```

```
UnitDefs[216]["showPlayerName"] = true,
UnitDefs[216]["slideTolerance"] = 0,
UnitDefs[216]["smoothAnim"] = true,
UnitDefs[216]["sonarJamRadius"] = 0,
UnitDefs[216]["sonarRadius"] = 300,
UnitDefs[216]["sounds"] = {
      ["repair"] = {
      }
      ["arrived"] = {
      }
      ["underattack"] = {
        [1] = {
          ["id"] = 140,
          ["name"] = "warning2",
          ["volume"] = 5,
        }
      }
      ["select"] = {
        [1] = {
          ["id"] = 213,
          ["name"] = "kccorsel",
          ["volume"] = 5,
        }
      }
      ["deactivate"] = {
      }
      ["activate"] = {
      }
      ["ok"] = {
        [1] = {
          ["id"] = 212,
          ["name"] = "kcormov",
          ["volume"] = 5,
        }
      }
      ["cant"] = {
        [1] = {
          ["id"] = 92,
          ["name"] = "cantdo4",
          ["volume"] = 5,
        }
      }
      ["working"] = {
      }
      ["build"] = {
        [1] = {
          ["id"] = 197,
          ["name"] = "nanlath2",
          ["volume"] = 5,
        }
      }
    }
UnitDefs[216]["speed"] = 37.5,
UnitDefs[216]["speedToFront"] = 0.070000000298023,
UnitDefs[216]["springCategories"] = {
      ["commander"] = false,
      ["notship"] = false,
      ["notair"] = false,
      ["all"] = false,
      ["notsub"] = false,
      ["weapon"] = false,
    }
UnitDefs[216]["startCloaked"] = false,
UnitDefs[216]["stealth"] = false,
```

```
UnitDefs[216]["stockpileWeaponDef"] = nil,
UnitDefs[216]["targfac"] = false,
UnitDefs[216]["techLevel"] = 0,
UnitDefs[216]["terraformSpeed"] = 300,
UnitDefs[216]["tidalGenerator"] = 0,
UnitDefs[216]["tooltip"] = "Commander",
UnitDefs[216]["totalEnergyOut"] = 25,
UnitDefs[216]["trackOffset"] = 0,
UnitDefs[216]["trackStrength"] = 0,
UnitDefs[216]["trackStretch"] = 1,
UnitDefs[216]["trackType"] = 0,
UnitDefs[216]["trackWidth"] = 32,
UnitDefs[216]["transportByEnemy"] = true,
UnitDefs[216]["transportCapacity"] = 0,
UnitDefs[216]["transportMass"] = 100000,
UnitDefs[216]["transportSize"] = 0,
UnitDefs[216]["transportUnloadMethod"] = 0,
UnitDefs[216]["turnRadius"] = 500,
UnitDefs[216]["turnRate"] = 1133,
UnitDefs[216]["type"] = "Builder",
UnitDefs[216]["unitFallSpeed"] = 1.0,
UnitDefs[216]["upright"] = true,
UnitDefs[216]["useBuildingGroundDecal"] = false,
UnitDefs[216]["useHitSphereOffset"] = false,
UnitDefs[216]["wantedHeight"] = 0,
UnitDefs[216]["waterline"] = 0,
UnitDefs[216]["weapons"] = {
     [1] = {
       ["onlyTargets"] = {
         ["antiflame"] = false,
         ["vtol"] = false,
         ["notland"] = false,
         ["fort"] = false,
         ["special"] = false,
         ["notair"] = false,
         ["kbot"] = false,
         ["antiemg"] = false,
         ["commander"] = false,
         ["jam"] = false,
         ["tport"] = false,
         ["constr"] = false,
         ["strategic"] = false,
         ["kamikaze"] = false,
         ["minelayer"] = false,
         ["hover"] = false,
         ["noweapon"] = false,
         ["plant"] = false,
         ["ship"] = false,
         ["antilaser"] = false,
         ["phib"] = false,
         ["mine"] = false,
         ["notstructure"] = false,
         ["tank"] = false,
         ["mobile"] = false,
         ["underwater"] = false,
         ["antigator"] = false,
         ["notship"] = false,
         ["all"] = false,
         ["notsub"] = false,
         ["weapon"] = false,
       }
       ["weaponDef"] = 39,
       ["slavedTo"] = 0,
       ["badTargets"] = {
```

```
          ["antilaser"] = false,
        }
        ["fuelUsage"] = 0,
        ["mainDirX"]  = 0,
        ["mainDirY"]  = 0,
        ["mainDirZ"]  = 1,
        ["maxAngleDif"] = -1,
      }
      [2] = {
        ["onlyTargets"] = {
        }
        ["weaponDef"] = 195,
        ["slavedTo"] = 0,
        ["badTargets"] = {
        }
        ["fuelUsage"] = 0,
        ["maxAngleDif"] = -1,
      }
      [3] = {
        ["onlyTargets"] = {
          ["antiflame"] = false,
          ["vtol"] = false,
          ["notland"] = false,
          ["fort"] = false,
          ["special"] = false,
          ["notair"] = false,
          ["kbot"] = false,
          ["antiemg"] = false,
          ["commander"] = false,
          ["jam"] = false,
          ["tport"] = false,
          ["constr"] = false,
          ["strategic"] = false,
          ["kamikaze"] = false,
          ["minelayer"] = false,
          ["hover"] = false,
          ["noweapon"] = false,
          ["plant"] = false,
          ["ship"] = false,
          ["antilaser"] = false,
          ["phib"] = false,
          ["mine"] = false,
          ["notstructure"] = false,
          ["tank"] = false,
          ["mobile"] = false,
          ["underwater"] = false,
          ["antigator"] = false,
          ["notship"] = false,
          ["all"] = false,
          ["notsub"] = false,
          ["weapon"] = false,
        }
        ["weaponDef"] = 13,
        ["slavedTo"] = 0,
        ["badTargets"] = {
        }
        ["fuelUsage"] = 0,
        ["maxAngleDif"] = -1,
      }
    }
UnitDefs[216]["windGenerator"] = 0,
UnitDefs[216]["wingAngle"] = 0.079999998211861,
UnitDefs[216]["wingDrag"] = 0.070000000298023,
UnitDefs[216]["wreckName"] = "CORCOM_DEAD",
```

```
UnitDefs[216]["xsize"] = 4,
UnitDefs[216]["ysize"] = 4,
UnitDefs[216]["canParalyze"] = false,
UnitDefs[216]["canStockpile"] = false,
UnitDefs[216]["hasShield"] = false,
UnitDefs[216]["canAttackWater"] = false,
UnitDefs[216]["cost"] = 2541.6667480469,
------ -- -
```

Game.armorTypes

This table is both indexed and keyed (bi-directional). This means if you have the index you can get the name, and vise-versa. Also be aware that many mods do not use the full range of types:

Index	Name	Meaning
0	`default`	Anything not defined specifically
1	`amphibious`	Amphibous tanks and bots
2	`anniddm`	Annihilator?
3	`antibomber`	
4	`antifighter`	
5	`antiraider`	
6	`atl`	Advanced Torpedo Launcher
7	`blackhydra`	Flagship
8	`commanders`	
9	`crawlingbombs`	
10	`dl`	Depthcharge launcher
11	`else`	Anything not covered?
12	`flakboats`	Naval anti-air
13	`flaks`	Anti-air flak
14	`flamethrowers`	
15	`gunships`	
16	`heavyunits`	Not sure how heavy a unit needs to need to qualify
17	`hgunships`	Heavy Gunships
18	`jammerboats`	
19	`krogoth`	Giant Core T3 bot
20	`l1bombers`	
21	`l1fighters`	
22	`l1subs`	
23	`l2bombers`	
24	`l2fighters`	
25	`l2subs`	
26	`l3subs`	
27	`mechs`	Kbots

28	minelayers	
29	mines	
30	nanos	
31	orcone	
32	otherboats	
33	plasmaguns	
34	radar	
35	seadragon	
36	spies	
37	tl	Torpedo Launcher
38	vradar	VTOL Radar Plane
39	vtol	General planes
40	vtrans	VTOL Transports
41	vulcbuzz	Large Multi-barrelled artilery (Vulcan and Buzzsaw)

Keysyms (Keyboard Input Codes)

A table named KEYSYMS provides contants for accessing keyboard input. It's important to remember that both the table name and the key names must always be uppercase. The table is defined in *LuaUI/Headers/keysym.h.lua* and must be imported before use with the following line (somewhere near the top of your widget file):

```
include("keysym.h.lua")
```

Name	Code	Comments
UNKNOWN	0	
FIRST	0	
BACKSPACE	8	
TAB	9	
CLEAR	12	
RETURN	13	
PAUSE	19	
ESCAPE	27	
SPACE	32	
EXCLAIM	33	!
QUOTEDBL	34	"
HASH	35	#
DOLLAR	36	$
AMPERSAND	38	&
QUOTE	39	'

LEFTPAREN	40	(
RIGHTPAREN	41	)
ASTERISK	42	*
PLUS	43	+
COMMA	44	,
MINUS	45	-
PERIOD	46	.
SLASH	47	/
N_0	48	See KP* codes for keypad numbers
N_1	49	
N_2	50	
N_3	51	
N_4	52	
N_5	53	
N_6	54	
N_7	55	
N_8	56	
N_9	57	
COLON	58	:
SEMICOLON	59	;
LESS	60	<
EQUALS	61	=
GREATER	62	>
QUESTION	63	?
AT	64	@
LEFTBRACKET	91	(
BACKSLASH	92	\
RIGHTBRACKET	93	)
CARET	94	^
UNDERSCORE	95	_
BACKQUOTE	96	
A	97	Actually 'a' ascii
B	98	
C	99	
D	100	
E	101	
F	102	

G	103	
H	104	
I	105	
J	106	
K	107	
L	108	
M	109	
N	110	
O	111	
P	112	
Q	113	
R	114	
S	115	
T	116	
U	117	
V	118	
W	119	
X	120	
Y	121	
Z	122	
DELETE	127	
WORLD_0	160	0xA0 (International keyboards only)
WORLD_1	161	0xA1 ...
WORLD_2	162	...
WORLD_3	163	
WORLD_4	164	
WORLD_5	165	
WORLD_6	166	
WORLD_7	167	
WORLD_8	168	
WORLD_9	169	
WORLD_10	170	
WORLD_11	171	
WORLD_12	172	
WORLD_13	173	
WORLD_14	174	
WORLD_15	175	

WORLD_16	176	
WORLD_17	177	
WORLD_18	178	
WORLD_19	179	
WORLD_20	180	
WORLD_21	181	
WORLD_22	182	
WORLD_23	183	
WORLD_24	184	
WORLD_25	185	
WORLD_26	186	
WORLD_27	187	
WORLD_28	188	
WORLD_29	189	
WORLD_30	190	
WORLD_31	191	
WORLD_32	192	
WORLD_33	193	
WORLD_34	194	
WORLD_35	195	
WORLD_36	196	
WORLD_37	197	
WORLD_38	198	
WORLD_39	199	
WORLD_40	200	
WORLD_41	201	
WORLD_42	202	
WORLD_43	203	
WORLD_44	204	
WORLD_45	205	
WORLD_46	206	
WORLD_47	207	
WORLD_48	208	
WORLD_49	209	
WORLD_50	210	
WORLD_51	211	
WORLD_52	212	

WORLD_53	213	
WORLD_54	214	
WORLD_55	215	
WORLD_56	216	
WORLD_57	217	
WORLD_58	218	
WORLD_59	219	
WORLD_60	220	
WORLD_61	221	
WORLD_62	222	
WORLD_63	223	
WORLD_64	224	
WORLD_65	225	
WORLD_66	226	
WORLD_67	227	
WORLD_68	228	
WORLD_69	229	
WORLD_70	230	
WORLD_71	231	
WORLD_72	232	
WORLD_73	233	
WORLD_74	234	
WORLD_75	235	
WORLD_76	236	
WORLD_77	237	
WORLD_78	238	
WORLD_79	239	
WORLD_80	240	
WORLD_81	241	
WORLD_82	242	
WORLD_83	243	
WORLD_84	244	
WORLD_85	245	
WORLD_86	246	
WORLD_87	247	
WORLD_88	248	
WORLD_89	249	

WORLD_90	250	
WORLD_91	251	
WORLD_92	252	
WORLD_93	253	
WORLD_94	254	
WORLD_95	255	0xFF
KP0	256	
KP1	257	
KP2	258	
KP3	259	
KP4	260	
KP5	261	
KP6	262	
KP7	263	
KP8	264	
KP9	265	
KP_PERIOD	266	
KP_DIVIDE	267	
KP_MULTIPLY	268	
KP_MINUS	269	
KP_PLUS	270	
KP_ENTER	271	
KP_EQUALS	272	
UP	273	
DOWN	274	
RIGHT	275	
LEFT	276	
INSERT	277	
HOME	278	
END	279	
PAGEUP	280	
PAGEDOWN	281	
F1	282	
F2	283	
F3	284	
F4	285	
F5	286	

F6	287	
F7	288	
F8	289	
F9	290	
F10	291	
F11	292	
F12	293	
F13	294	
F14	295	
F15	296	
NUMLOCK	300	
CAPSLOCK	301	
SCROLLOCK	302	
RSHIFT	303	
LSHIFT	304	
RCTRL	305	
LCTRL	306	
RALT	307	
LALT	308	
RMETA	309	
LMETA	310	
LSUPER	311	Left "Windows" key
RSUPER	312	Right "Windows" key
MODE	313	Alt Gr key
COMPOSE	314	Multi-key compose key
HELP	315	
PRINT	316	
SYSREQ	317	
BREAK	318	
MENU	319	
POWER	320	Power Macintosh power key
EURO	321	Some european keyboards
UNDO	322	Atari keyboard has Undo
LAST	323	

VFS Modes

These modes are constants passed to VFS.Include() and friends to control the order that archives are searched for a given file. They are case-sensitive.

VFS.RAW	Only select uncompressed files.
VFS.MOD	(unimplemented, acts like VFS.ZIP)
VFS.MAP	(unimplemented, acts like VFS.ZIP)
VFS.BASE	(unimplemented, acts like VFS.ZIP)
VFS.ZIP	Only select compressed files (.sdz,.sd7).
VFS.RAW_FIRST	Try uncompressed files first, then compressed.
VFS.ZIP_FIRST	Try compressed files first, then uncompressed.
VFS.RAW_ONLY	Deprecated. Same as VFS.RAW.
VFS.ZIP_ONLY	Deprecated. Same as VFS.ZIP.

MORE INFORMATION

Spring Website

http://springrts.com/

Official Lua language documentation

'Programming in Lua' book online
Lua 5.1 Reference manual
Lua tutorials

Spring IRC Channels

Type /j #channelname in Spring lobby.

#lua - Spring Lua questions and discussion

#sy - Spring engine development discussion

CREDITS

Since documentation on the Lua scripts is very hard to find I have drawn on a lot of information provided by other developers through code comments, forums and IRC/lobby channels. Wherever possible I have listed here their screen names and a rough outline of how they were helpful.

SpliFF
Me. Metalstorm mod developer. Compiled this document from many sources.

Hoi
Helped me get started.

Evil4Zerggin
Gadgets overview based on forum post.

FLOZi

Spring and 1944 mod developer and provider of useful forum responses.

lurker
Helped me with questions I had about the LLTA 'Kills for Tech' gadget and general tips. Lurks in #lua

KDR_11k
Spring developer. AllowCommand/CommandFallback descriptions.

imbaczek
Spring Developer. AllowCommand/CommandFallback clarifications.

Argh
AllowCommand/CommandFallback examples.

Clogger
Definition of LuaGaia and COB and the difference between widgets and gadgets.

Trepan
Spring developer. Comments in LuaREADME.

jK
Spring developer. UnitDef, WeaponDef and FeatureDef tables. Function defs on wiki. Function access list. Info on planned event system changes. Performance tests. Provided most of the Lua documentation on the Spring wiki.

Tobi
Spring developer. Wrote the unit script system and initial documentation.

quantum
Spring and CA Dev. Group AI info. DLL access via package.