HOMEWORLD2

# SCAR – Scripting at Relic


**SCAR – SCRIPTING AT RELIC ..... 1**

**1 A LESSON IN SCAR ..... 1**

**2 EVENT TABLES ..... 3**

***2.1 Creating Events ..... 4***

**3 UNIQUE FUNCTIONS ..... 6**

**4 CREATING A MISSION ..... 6**

***4.1 Creating the campaign structure ..... 6***

4.1.1 Creating a .level file ..... 6

***4.2 Creating the .lua file ..... 7***

4.2.1 Sample Mission.lua Script ..... 7

**5 SETTING UP THE .CAMPAIGN FILE ..... 9**

**6 MISCELLANEOUS FILES ..... 10**

***6.1 .DAT FIles ..... 10***

6.1.1 Localizing your mission ..... 10

***6.2 Teamcolour.lua ..... 10***

***6.3 AI.lua ..... 11***

***6.4 ReferenceFleet.lua ..... 11***

***6.5 Datfiles.lua ..... 11***

***6.6 Mission.tga ..... 11***

**7 APPENDIX ..... 11**


## 1 A lesson in SCAR

SCAR (SCripting At Relic) is a linear scripting language utilizing a Rule based system, similar to the trigger based system that was used on ICPC but without the GUI. SCAR is based on the scripting language LUA, and contains much of its inherent functionality (such as functions, conditional statements, loops, etc). Access to the game's systems are provided via functions that

the programmers create, that enable us to determine that status of certain entities, as well as issue orders, and modify the stats of those entities.

A Rule is simply a user-defined function that, once "added", is then evaluated by the game every interval. The Rule is added by using the `Rule_Add( "name of Rule" )` function, and passing in the name of the Rule that you wish to add. The interval that the Rule is evaluated at can either be the default (every frame) or can be set (every x seconds), using the `Rule_AddInterval` function in place of the `Rule_Add`. The interval can be defined on a per-Rule basis, allowing you to specify a different interval for every Rule that you have.

For example, if we wanted a Rule that printed out "Hello world!" to the console every frame, we would have something along the lines of:

```
-- add the Rule_HelloWorld to the queue of Rules being evaluated
Rule_Add( "Rule_HelloWorld" )

-- here the Rule_HelloWorld function is defined, in the LUA standard
function Rule_HelloWorld()
	print( "Hello world!" )
end
```

This will then print the words "Hello world!" to the console every frame.

Once a Rule has been added, every interval the contents of that Rule will be executed by the game. If the first statement in a Rule is a conditional, such as an `if` statement, this will ensure that the Rule is only executed once the conditions are met.

Lets use a conditional statement to print the words "Hello World!" only when the player is of the race Hiigaran.

```
Rule_Add( "Rule_HelloWorld" )

function Rule_HelloWorld()
	-- use the Player_GetRace function to determine if player 0 is Hiigaran
	if ( Player_GetRace( 0 ) == Race_Hiigaran ) then
		print( "Hello world!" )
		Rule_Remove( "Rule_HelloWorld" )
	end
end
```

Notice the call that's being made to `Rule_Remove`. This removes the Rule from the queue, meaning that it will not be evaluated on the next interval. If you don't remove the Rule after it has been used, it will continue to be evaluated every specified interval. By adding the `Rule_Remove`, we ensure that this Rule will only be executed once whenever it's conditional statement is met.

A Rule can be removed at any stage during your script, but it will not be completely removed until the next Interval. So if you remove a Rule in the middle of the Rule, the remaining calls of that Rule will still be executed in this Interval.

This looks good, but what happens if the player's ID isn't 0, or perhaps there are multiple players and we want to check each one? We can wrap our conditional statement in a `for` loop, which will then go through every player in the game and check to see if they are Hiigaran or not.

```
Rule_Add( "Rule_HelloWorld" )

function Rule_HelloWorld()
	-- we use the for loop, an functionality provided by Lua
	for i=1,Universe_PlayerCount() do
```

```
                if ( Player_GetRace( i ) == Race_Hiigaran ) then
                        print( "Hello world!" )

                end
                -- once all players are checked, remove this rule
                if ( i == Universe_PlayerCount() ) then
                        Rule_Remove( "Rule_HelloWorld" )
                end
        end
end
```

In this Rule, the `for` loop checks every player in the game to see if they are Hiigaran, printing "Hello World!" if they are. When the loop iterator (`i`) is equal to the number of players, the Rule is removed.

See how quickly a very simple Rule can turn into a very useful one? This is the inherent power of SCAR. At its most basic level, it can be quite functional. However by applying some more advanced scripting techniques, it will quickly become a powerful and versatile tool.

As you can see, a Rule is simply a function. The only thing that makes it a Rule is we append it with `Rule_` (this is simply a naming convention employed for clarity) and the fact that it is "added" to the list of Rules being evaluated as opposed to called directly from the script. This means that you can also create simple functions inside your script to enable you to perform repetitive functions, instead of needing to have dozens of Rules all performing the same function.

## 2 Event Tables

Another feature of SCAR is the Event Table. An Event Table is a list of items that occur using a defined amount of time in order to determine when to step from one segment of the list to the next. They are purely linear, but can contain almost any function that you can call from a Rule. The only element not allowed within an Event is a conditional statement, such as "if" or a loop such as "for".

Events are most typically used for IntelEvents or Autofocuses, where you want to have a series of things occur on a controlled and linear sequence. The following is an example Event.

```
-- create the events table
Events = {} -- the name of this table must always be Events - this is what the game looks

-- here we name our event
Events.intelevent_constructinterceptors =
{
   {
        -- this is the first segment of the event
        -- we do some basic setup here, enabling universe skipping, turning on letterbox
        { "Universe_EnableSkip(1)", "" },
        { "Sound_EnterIntelEvent()","" },
        { "Sound_SetMuteActor('Fleet')", ""},
        HW2_Letterbox( 1 ),
        HW2_Wait(2),
   },
   {
        -- go into sensors manager, create a circle around the fighterproduction
        { "Sensors_EnableCameraZoom( 0 )","" },
        { "Sensors_Toggle( 0 )", "" },
        { "g_pointer_default = HW2_CreateEventPointerSubSystem( 'FighterProduction',
'Mothership' )", "" },
        { "Camera_Interpolate( 'here', 'camera_focusOnFighterSub', 2)",""},
        HW2_SubTitleEvent( Actor_FleetCommand, "$40550", 5  ),
   },
```

```
    {
        -- Fleet command is talking..again!
        HW2_SubTitleEvent( Actor_FleetCommand, "$40551", 5  ),
    },
    {
        HW2_Wait(1),
    },
    {
        -- remove the circle, create a primary objective
        { "EventPointer_Remove(g_pointer_default)", "" },
        { "obj_prim_buildtwointerceptors_id = Objective_Add(
obj_prim_buildtwointerceptors, OT_Primary )", "" },
        { "Objective_AddDescription( obj_prim_buildtwointerceptors_id, '$40965')", "" },
        { "Objective_AddDescription( obj_prim_buildtwointerceptors_id, '$40966')", "" },
        { "Player_UnrestrictBuildOption( g_playerID, 'Hgn_Interceptor' )",""},
        HW2_SubTitleEvent( Actor_FleetIntel, "$40552", 5 ),
    },
    {
        -- disable letterbox, general cleanup of the event
        HW2_Letterbox( 0 ),
        HW2_Wait(2),
        { "Sound_SetMuteActor('')", ""},
        { "Sound_ExitIntelEvent()","" },
        { "Sensors_EnableCameraZoom( 1 )","" },
        { "Universe_EnableSkip(0)", "" },
    },
}
```

Events are called using the Event_Start() function. The Event_Start function is passed the name of the Event that you wish to play. The Event that you have called will then begin playing from start to finish. So, lets see how this is incorporated into our HelloWorld rule from above. We will replace our effectively useless print statement with a call that actually starts playing the above Event.

```
Rule_Add( "Rule_HelloWorld" )

function Rule_HelloWorld()
        -- we use the for loop, an functionality provided by Lua
        for i=1,Universe_PlayerCount() do
                if ( Player_GetRace( i ) == Race_Hiigaran ) then
                        -- instead of printing Hello World, start the event
                        Event_Start( "intelevent_constructinterceptors" )
                end
                -- once all players are checked, remove this rule
                if ( i == Universe_PlayerCount() ) then
                        Rule_Remove( "Rule_HelloWorld" )
                end
        end
end
```

Events do look complicated, but actually they are very straightforward. Let's break it down into its component parts, and gain a better understanding of the overall picture.

## *2.1 Creating Events*

The very first thing that's done is initializing the events table. This is done with:

```
Events = {}
```

This must always be called at the start of your Events, only be called once, and must always be called "Events", as that's the specific name that the game will look for. Without this, all your

Events will not work, so be sure to check for the existence of this if you start running into problems.

The next step is naming our Event. This is done with the following:

```
Events.intelevent_constructinterceptors =
```

So here we have an Event called “intelevent_constructinterceptors”. We use this name later in order to begin playing the Event and for checking when it’s complete. For this reason, Event names must be unique.

Now we get to the meat of the Event, the actual block of function calls that determine what is going to occur. As you can see, the table is broken up into segments, each contained within the curly braces { and }. Anything that is called inside these segments is executed at the same time. So the following:

```
{
{ "Universe_EnableSkip(1)", "" },
{ "Sound_EnterIntelEvent()","" },
{ "Sound_SetMuteActor('Fleet')", ""},
HW2_Letterbox( 1 ),
HW2_Wait(2),
},
```

will turn on the enable skipping, tell the sound we are entering an Intelevent, stop Fleet Command from talking, enable Letterbox, and Wait for 2 seconds. The key is the `HW2_Wait( 2 )`. This acts as the Controller. The Controller determines when we will move to the next segment of the Event list, in this case, we will Wait 2 seconds.

Events are separated into distinct segments so that we can pause at the current segment for a period of time, allowing the Events of the current segment to complete before beginning the items in the next.

As you can see, the contents of a segment are simply function calls. You will notice a difference between the call to say, Universe_EnableSkip(1) and the HW2_Wait(2) function. One is wrapped in further curly braces, and has a second empty item, whereas the other is a call to a function.

The distinction between the two is that HW2_Wait is a helper function the returns the following:

```
{ “wID = Wait_Start( 2 )”, "Wait_End( wID )” },
```

notice how the second element of the list now contains another function call? The second element specifies the function that, when it returns true, ensures this Controller will step to the next segment of the Event list.

In the above example, the first item is a call to Wait_Start, which returns a unique ID. The second item calls Wait_End with that unique ID as a parameter. When the specified amount of time has passed, Wait_End will return true. This will cause this Controller to allow the segment to step to the next.

If the second item is empty, it means that the first function will simply execute and that this item will not act as a Controller.

So if we were to write this segment without the helper functions, it would look like so:

```
{
    { "Universe_EnableSkip(1)", "" },
    { "Sound_EnterIntelEvent()","" },
    { "Sound_SetMuteActor('Fleet')", ""},
```

```
        { "Camera_SetLetterboxStateNoUI( 1, 2 )", "" },
        { "wID = Wait_Start( 2 )", "Wait_End( wID )" },
    },
```

# 3 Unique Functions

If you've taken a look at the SCAR scripts for the missions in Homeworld2, you will notice some commonalities, such as the `OnInit()` function.

The `OnInit()` function is run whenever the mission is first initialized, and generally contains a call to add the `Rule_Init()`. The different between `OnInit()` and `Rule_Init()` is that the game will look for an `OnInit()` function when it loads the mission. This is the entry point, what the game uses to get your mission started. If there is nothing inside the `OnInit()` function, then your mission wont do anything!

It's worth noting the distinction between the `OnInit()` function and say, the `OnStartOrLoad()` function. `OnInit()` will only run when the mission is loaded for the first time. If the mission is loaded from a save game, this function will not be run. Hence we have the `OnStartOrLoad()`. If you have anything that you need to be initialized even from a save game (such as looping effects), include the calls in your `OnStartOrLoad()` function

# 4 Creating a Mission

Here we will outline the basic steps required in order to create a mission for Homeworld2. We will first create a new campaign for the purposes of this tutorial, and then create the first, basic mission for the campaign.

Once the directory structure is properly setup, there are 3 steps to creating a mission. First, create and export the mission .level file, secondly create a basic .lua file, and thirdly add this mission to the .campaign file.

In order to run through this step-by-step process, you will need:

A copy of Maya installed and setup with the Relic Toolshelf properly installed.

A copy of Homeworld2 installed (you will need the source data, not the .big files).

## *4.1 Creating the campaign structure*

Firstly we will need to create the directory structure that will accommodate our new Campaign.

All actual missions in a campaign are stored in the data\LevelData\Campaign folder. Within this folder there are a number of subdirectories, all of which contain more subdirectories that are where the actual mission files are stored. We will now create a "Postmortem" campaign.

First, create a folder in the data\LevelData\Campaign directory called "Postmortem". This will be the name of our new campaign (not very imaginative, but suits our purposes). Here is where all of our missions will be stored.

You have probably noticed that there are a number of .campaign files in the Campaign folder. These define what missions play where in our campaign, along with things such as the animatics etc. We will get to these files later.

### 4.1.1 Creating a .level file

Now that we have a basic Campaign structure started, we need to create some actual .level files for our missions. To create a .level file, start up Maya, and open the LevelEd tool. The details of

the LevelEd tool are deep, and are covered in a separate tutorial. The basic elements that your level will require in order to make it functional are:

- A player start point, if this level is using persistent fleet. If not, it will require a Mothership that belongs to Player 0 (for more information on how to create and use persistent fleet see the Reactive Fleet postmortem).
- Ships for the opposition (any ships that belong to a player other than Player0 or Galaxy).
- Place resources on the map.
- Set the Race ID for any of the players that you have created.
- Set the Background for the mission.

Seeing as we are creating the first mission in our Campaign, place the following items into your .level file:

- An Hiigaran Mothership that belongs to Player0, and belonsg to a SobGroup called “Player_Mothership”
- A Vaygr Carrier that belongs to Player1, and belongs to a SobGroup called “AI1_Carrier”
- Set the race for Player0 to Hiigaran, and the race for Player1 to Vaygr.
- Set the Background to M01

Once you have placed these items on your map, create a new directory in datasrc\LevelData\Campaign\Postmortem\Mission_01, and save the file as a .ma file into the Mission_01 folder with the same name (so Mission_01.ma).

Now export the file into a new directory within your “Postmortem” Campaign directory called Mission_01, and call the .level file Mission_01.level. Your .level file will now be stored in data\LevelData\Campaign\Postmortem\Mission_01.

## *4.2 Creating the .lua file*

Now that you have your basic mission level file, you need to create your mission script. The .lua file is the main mission script. It defines all the functions and events in your mission.

The .lua file for the mission is where all of the Rules that will be in this mission. The .lua file must be named the same as the .level file, so in our case, this will be Mission_01.lua. It must also be stored in the same directory.

### 4.2.1 Sample Mission.lua Script

The following is a basic mission .lua file:

```
-- import library files, this includes all the helper functions
dofilepath("data:scripts/SCAR/SCAR_Util.lua")

-- variables can be created outside of everything, giving them global scope
obj_prim_newobj_id = 0

-- call the OnInit, otherwise nothings going to run
function OnInit()
	-- add the Rule_Init
	Rule_Add("Rule_Init")
	-- because OnInit isn't a Rule, we don't need to remove it
end

-- for standardization, we always have a Rule_Init to start things off
function Rule_Init()
```

```
        -- here we can call the first Intelevent for the mission
        Event_Start( "intelevent_intro" )

        -- add the rules that determine mission flow
        Rule_Add( "Rule_PlayerLoses" )

        Rule_Add( "Rule_PlayerWins" )

        -- be sure to remove the Rule_Init
        Rule_Remove( "Rule_Init" )
End

-- this rule checks to see if the players mothership is destroyed
function Rule_PlayerLoses()
        if ( SobGroup_Empty( "Player_Mothership" ) == 1 ) then

                -- fail our objective
                Objective_SetState( obj_prim_newobj_id, OS_Failed )

                -- player loses the mission
                setMissionComplete( 0 )

                Rule_Remove( "Rule_PlayerLoses" )
        end
end

-- this rule checks to see if the player has destroyed the enemy carrier
function Rule_PlayerWins()
        if ( SobGroup_Empty( "AI1_Carrier" ) == 1 ) then

                -- complete our objective
                Objective_SetState( obj_prim_newobj_id, OS_Complete )

                -- this means player wins mission
                setMissionComplete( 1 )

                Rule_Remove( "Rule_PlayerWins" )
        end
end

-- don't forget to create the Events table!
Events = {}

-- this is the intro intelevent
Events.intelevent_intro =
{
   {
        { "Sound_EnableAllSpeech( 1 )",""},
        { "Sound_EnterIntelEvent()","" },
        { "Universe_EnableSkip(1)", "" },
        HW2_LocationCardEvent( "Postmortem Tutorial", 5 ),
   },
   {
        HW2_Letterbox( 1 ),
        HW2_Wait( 2 ),
   },
   {
        HW2_SubTitleEvent( Actor_FleetCommand, "This is a tutorial script", 5 ),
   },
   {
        HW2_Wait( 1 ),
   },
   {
        { "obj_prim_newobj_id = Objective_Add( "A new Objective", OT_Primary )", "" },
        { "Objective_AddDescription( obj_prim_startresourcing_id, 'Description')", "" },
        HW2_SubTitleEvent( Actor_FleetIntel, "Ive just issued a new objective", 4 ),
```

```
    },
    {
        HW2_Wait(1),
    },
    {
        HW2_Letterbox( 0 ),
        HW2_Wait(2),
        { "Universe_EnableSkip(0)", "" },
        { "Sound_ExitIntelEvent()","" },
    },
}
```

# 5 Setting up the .campaign file

You will now need to create a campaign file for your new campaign. A campaign file indicates where the missions for this campaign are stored, their order, as well as information for animatics and such.

For your “Postmortem” campaign, create a Postmortem.campaign file in the data\LevelData\Campaign directory. The contents of this file should be as follows:

```
-- ============================================================================
--     Name    : Postmortem.campaign
--     Purpose : campaign for postmortem tutorial.
--
--     Copyright Relic Entertainment, Inc.  All rights reserved.
-- ============================================================================

-- DAT strings found in UI.DAT

-- localized display name for the UI
displayName = "Postmortem"

-- initialization
Mission = { }   -- create a mission structure

-- Mission 1
Mission[1] = {

    postload   = function () playAnimatic("data:animatics/A00.lua",1,1);  end,
                -- This next fucntion tells the game to play Animatic A00.lua before the
                mission loads.
    directory  = "Mission_01",
                -- This tells the scrtip what Directory to load the mission from. In this
                case from Data/Leveldata/Campiagn/Postmortem /Mission_01.
    level      = "Mission_01.level",
                -- This tells the script what .lua file to load
    postlevel  = function ( bWin ) if ( bWin == 1 ) then
playAnimatic("data:animatics/A01.lua", 1, 0) else postLevelComplete() end end,
                -- This tells the game what happens when the mission ends
    displayName    = "Mission 1",
                -- Used in the Campaign Description Menu
    description      = "Mission 1",
                -- Used in the Campaign Description Menu
}
```

When you add a new mission, you will need to add another Mission entry to this file.

# 6 Miscellaneous Files

Some other, extra steps that you may wish to do are setting up Localization, adding AI specific scripts, creating a loading screen and setting up the teamcolor for each player in the mission.

All these files are located in the Mission Folder and include: .dat files. Teamcolour.lua, AI.lua, ReferenceFleet.lua, datfiles.lua, Mission.tga)

## *6.1 .DAT FIles*

Localized text. Using ID Strings, the game matches the localization call ($42999) with the associated text as defined in the .dat file. For example:

```
42999 This is the localized text that would appear in the game.
```

And this is the Event Call that queries for this text:

```
    {
        HW2_SubTitleEvent( Actor_FleetCommand, "$42999", 5 ),
    },
```

In this case, the script will display the Actor Fleet Command and show the subtitle "`This is the localized text that would appear in the game`". The 5 shows how long the text remains on screen. In this case 5 seconds.

If you are using audio files, the game will match the subtitle dat number with the corresponding named audio file in `Data\Sound\english\Speech\MISSIONS`. Both the audio file and the dat number must be the same. In this case the audio file for 42999 would be 42999.fda

### 6.1.1 Localizing your mission

To localize your mission, you will need to add a similar campaign structure to the data\Locale\English\LevelData\Campaign\. In there you will need to place a .dat file with the proper formatting, that contains all the appropriate text for your mission.

Next, you will need to create a datfiles.lua file, that is stored in the same directory as your mission.level file. This will look something like:

```
Dictionaries =
{
        {
                name = "locale:leveldata/campaign/postmortem/mission_01.dat",
        },
}
```

This ensures the game will look to the correct directory and file for your localized text.

## *6.2 Teamcolour.lua*

This file specifies the base and stripe colour for all the players in the mission. Badges can also be specified here.

To customize the teamcolor of the players in your mission, create a teamcolor.lua file and place it in the same folder as your mission. An example of the contents of the teamcolor.lua file is as follows:

```
-- [Player Index] = {{Teamcolour R, G, B}, {Stripecolour R, G, B}, "BadgeFileName.tga"),

teamcolours =
{
```

```
[0] =
{{.365,.553,.667},{.800,.800,.800},"DATA:Badges/Hiigaran.tga",{.365,.553,.667},"data:/eff
ect/trails/hgn_trail_clr.tga"}, -- player
[1]
={{.900,.900,.900},{.100,.100,.100},"DATA:Badges/Vaygr.tga",{.921,.75,.419},"data:/effect
/trails/vgr_trail_clr.tga"} -- vaygr
}
```

### 6.3 AI.lua

AI script. This overrides or activates AI functions. To add specific AI to your missions, you need to add an AIX.lua in your mission directory, where X is the ID number of the AI you wish to customize.

The actual contents of the AIX.lua file are beyond the scope of this document.

### 6.4 ReferenceFleet.lua

Used to determine AI difficulty. The mission will need reactive fleet slots added to it.

### 6.5 Datfiles.lua

Specifies all the dat files used by the mission.

### 6.6 Mission.tga

Loading screen used when, you guessed it, loading a mission. The loading screen is a .tga file that exists in the same directory as the .level file, and has the same name.

## 7 Appendix

- For further information on LUA, refer to the online documentation at http://www.lua.org.
- Recommended program for editing a Lua file? Get Scite at http://www.scintilla.org/SciTE.html.