# Ability Manager Framework

User manual

Author : X3N0-Life-Form

Version : 1.0

Created on : 05/11/16

# Table of Contents

# Quick start – the abridged version

## Required files

- data/scripts
  - parse.lua
  - abilityManager.lua
  - abilityLibrary.lua *(--> new ability templates go there)*
  - shipVariant.lua
  - shipVariantMissionWide.lua
- data/tables
- data/config
  - abilities.tbl
  - ship_variants.tbl *(--> not actually related to abilities (for now))*
  - ship_variants_mission_wide.tbl *(aka SVMW.tbl)*

## SEXP calls

- Set abilities for everybody = *setShipVariant('some category name from SVMW.tbl')*
- Set ability for someone = *ability_attachAbility(className, shipName, isManuallyFired)*
- Manual trigger = *ability_trigger(' AWAXEM::DISCO FURY')*

# Mod setup

## Lua Scripts

First off, you need to set up the scripts the framework relies on. Step one is putting the following *.lua files in *<your mod>/data/scripts* :

- **parse.lua :**
  - Parses the various *.tbl files used by the other scripts.
- **abilityManager.lua :**
  - The heart of the framework, this is the file that handles everything ability-related.
  - There's a number of so-called "high-level" functions that may be of interest to modders, more on that later.
- **abilityLibrary.lua :**
  - Contains a number of sample abilities to play with.
- **shipVariant.lua** and **shipVariantMissionWide.lua** :
  - These help setting up ship abilities without the need to manually attach abilities to ships.

Note : Each of these scripts is prefaced with a description of what they do and how to use them.

Step two is loading each of these scripts with a modular scripting table (see *AMD-script_init-sct.tbm*) :

```
#Conditional Hooks
$Application: FS2_Open

;; Master script
$On Game Init: [[parse.lua]]

;; Ship Variant
$On Game Init: [[shipVariant.lua]]
$On Warp In: [setVariantDelayed()]

$On Game Init: [[shipVariantMissionWide.lua]]

;; Ability Management
$On Game Init:[[abilityManager.lua]]
$On Gameplay Start:[ability_resetMissionVariables()]
$On Game Init:[[abilityLibrary.lua]]

$State: GS_STATE_GAME_PLAY
$On Frame:[ability_cycleTrigger()]

#End
```

**Explaination :**

- The *.lua files themselves get loaded when the game starts, and those that require a config table also parse that table.
- When a mission starts, the AMF needs to reset a number of variables, so *resetMissionVariables()* gets called as the gameplay phase starts.
- During gameplay, the *cycleTrigger()* function gets called every frame, but only runs its payload every 0.1 seconds (see the *ability_castInterval* variable in abilityManager.lua). It cycles through every active ability in-mission and fires them if possible.
- *setVariantDelayed()* sets variants for any ship warping in during gameplay.

## Config tables

In order to work, the framework requires a number of config files, designed to emulate the structure of a standard FSO table :

| **Script** | **Table** |
|---|---|
| abilityManager.lua<br>abilityLibrary.lua | abilities.tbl |
| shipVariant.lua | ship_variants.tbl |
| shipVariantMissionWide.lua | ship_variants_mission_wide.tbl |

## abilities.tbl

Let's leave the variant-related table for now and focus on **abilities.tbl**. Below is a brief specification of the table's structure, fields and expected values. Most of these should be self-explainatory, but two attributes are of particular importance :

```
#Abilities
	$Name: string
	$Function: string
		* function to call when firing the ability
	$Target Type: list of string
		* ship types
	$Target Team: list of string
		* hostile, friendly, relative to the caster
	$Target Selection: list of string
		* Closest, Current target, Random, Self
	$Range: integer (optional)
	$Cost: integer/list of integer (tied to diffculty level) (optional)
		+Cost type: string (optional)
			* what ammo consumption system is to be used
			* Ammo (default), energy:weapon, energy:shield, energy:afterburner
		+Starting Reserve:	integer (optional)
	$Cooldown: number
		* ability cooldown time
	$Ability Data:
		* sub-attributes contain metadata used by the function called
#End
```

- The *$Function* field refers to the name of the function to call when the ability is being fired, as defined in **abilityLibrary.lua**.
- *$Ability Data* contains any data the ability function requires in order to run.

These two fields allow you to define a number of abilities with similar effects based on the same template. Let's look at the SSM strikes defined bellow, one of them is meant to be called by a capital ship, will target the closest target with no range limit, and its cooldown time is dependent on difficulty. The other is meant to be called by a fighter, and thus has more limitations, such as a finite range, a limited number of shots and a less powerful strike type.

Note that both use the same function, and have the same sub-attribute under *$Ability Data*.

```
$Name:                  SSM-moloch-std
$Function:              fireSSM
$Target Type:           cruiser, capital, corvette
$Target Team:           Hostile
$Target Selection:      Closest
$Cooldown:              24, 20, 17, 15, 13
$Ability Data:          fireSSM
  +Strike Type:         Shivan SSM Strike
```

```
$Name:                  Fighter-launched Recursive SSM
$Function:              fireSSM
$Target Type:           cruiser, capital, corvette
$Target Team:           Hostile
$Target Selection:      Current Target
$Range:                 7500
$Cost:                  1
    +Starting Reserve:  15
$Cooldown:              3
$Ability Data:
  +Strike Type:         Mordiggian Strike
```

## ship_variants_mission_wide.tbl

In order to make it easier to attach abilities to ships, I have modified the variant management script to do that for me. Essentially, this allows mission designers to set any number of ability to any number of ships using a single SEXP. The table itself is fairly straightforward :

- Each table category defines a variant package
- Each entry defines what variant and/or abilities are associated with a ship
    - The **+Manuel** sub-attribute specifies whether the abilities listed above are to be fired manually or not. In the example below, Virgo 1 has 4 abilities attached to it, with each needing to be fired manually.

```
#Demo 1

$Name:      Sobek
$Variant:   Standard

$Name:      Aeolus
$Variant:   Upgraded

$Name:      Moloch
$Variant:   Carrier Corvette
$Abilities: SSM-moloch-std

$Name:      Virgo 1
$Abilities: Energy Drain, Fighter-launched Mordiggian Strike, Repair-Self, Shield-Recharge-Target
    +Manual:true, true, true, true

$Name:      Virgo 2
$Abilities: Fighter-launched Mordiggian Strike

$Name:      Virgo 3
$Abilities: Fighter-launched Mordiggian Strike


$Name:      Repair Beacon
$Abilities: Repair-Field

#End
```

## ship_variants.tbl

Closely related to the previous table, it defines ship variants. Having no relation with the Ability Management Framework, describing the table is outside the scope of this document. You can find more info on this table, you can have a look at the description in the header of *shipVariant.lua*.

If you have no need for ship variants, you can leave this table empty.

# Mission design

## Under the hood – classes and instances

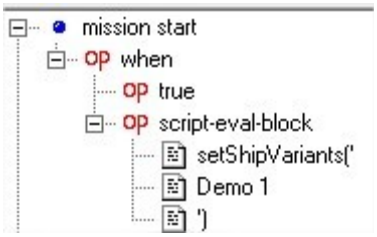
Before proceeding further, here is a brief description of how the framework handle abilities under the hood :

- Entries in abilities.tbl get converted into classes, identified by their **name**
- When attaching an ability to a ship, these classes are instanciated : an object referencing the names of the class and ship it belongs to is created.
  - An instance is identified by its **instance id**.
  - An instance id is generated like this **[ship name]::[ability class name]**. For instance, *AWAXEM::DISCO FURY* refers to an instance of the "DISCO FURY" ability, attached to the ship "AWAXEM".
- During a firing cycle, the framework goes through all instances, looks for a valid target in range according to its targetting heuristics, then if that ability can be fired, calls the relevant function and updates the instance's status.

## Attaching abilities to ships

As I mentionned in the config file section, I repurposed a variant management system to do the whole *attach-ability-to-ship* process for me, since I didn't ant to make a zillion SEXP calls to the *ability_attachAbility()* function.

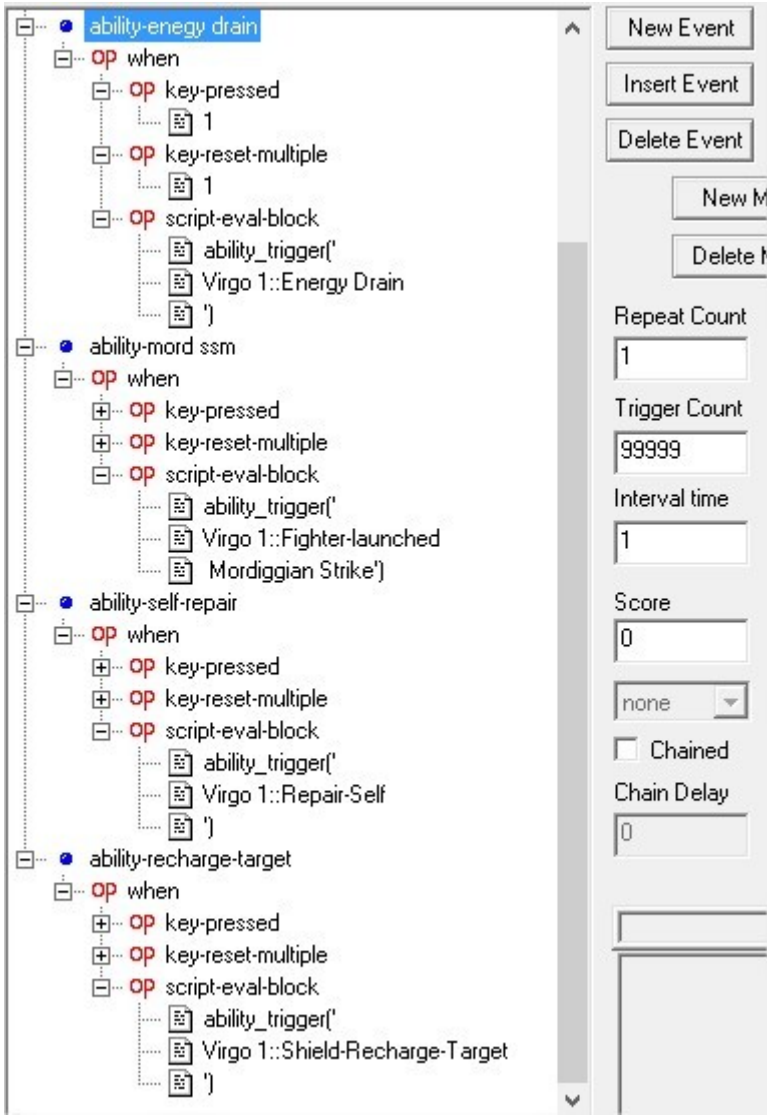
This means that the whole process only takes a single *script-eval* SEXP at mission start. The function that does all this is called **setShipVariants()**, and takes one argument – the name of one of the category defined in **ship_variants_mission_wide.tbl**, and then instanciates every ability defined there.



## Manually triggered abilities

Sometimes, you may want fire an ability manually through SEXPs rather leaving that to the framework. For that purpose, **ability_trigger(instanceId)** triggers a firing cycle (see "Under the hood" above).

Here is an example of four player-triggered abilities :



# Creating your own abilities

## Writing the function

Right now, ability functions need to have the following signature : **functionName(instance, class, targetName)**, where *instance* and *class* are the objects firing the ability. Notable things this gives you access to :

- The name of the ship firing the ability, through **instance.Ship**.
- The name of the ability's target.
- Values defined in the *$Ability Data* field, through **class.AbilityData['<field name>']**

```
function fireRepair(instance, class, targetName)
    local castingShip = mn.Ships[instance.Ship]
    local targetShip = mn.Ships[targetName]

    local hits = class.AbilityData['Hull']
    local shields = class.AbilityData['Shields']
    local weapons = class.AbilityData['Weapons']
    local afterburners = class.AbilityData['Afterburners']

    -- Note : don't repair things if they are dying
    if not (hits == nil) and not (targetShip.HitpointsLeft <= 0) then
        targetShip.HitpointsLeft = targetShip.HitpointsLeft + hits
        -- Make sure we don't go above 100%
        if (targetShip.HitpointsLeft > targetShip.HitpointsMax) then
            targetShip.HitpointsLeft = targetShip.HitpointsMax
        end
    end
```

```
$Name:                  Repair-Field
$Function:              fireRepair
$Target Type:           fighter, bomber
$Target Team:           Friendly
$Target Selection:      Random
$Range:                 900
$Cooldown:              1
$Ability Data:
    +Hull:          15
    +Shields:       150
```