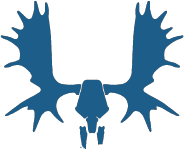
MOOSE User Guide



MMISSION OBJEOCT ORIENTOED SCRIPTISNG ENVIROENMENT

By Pikey and Habu v1.0

Table of Contents

[Document Purpose and Aim 1](#_Toc44145736)

[The Very Basics 1](#_Toc44145737)

[What is a Script? 1](#_Toc44145738)

[What is MOOSE? 1](#_Toc44145739)

[What Can MOOSE Do For Me? 1](#_Toc44145740)

[What If I Don’t Want To Learn Scripting? 2](#_Toc44145741)

[How Do I Get and Install MOOSE? 2](#_Toc44145742)

[How Do I Install MOOSE Scripts? 3](#_Toc44145743)

[What Next? 3](#_Toc44145744)

[Using Existing MOOSE Script Modules 4](#_Toc44145745)

[Where Do I Find MOOSE Scripts? 4](#_Toc44145746)

[How Do I Customize a Script? 4](#_Toc44145747)

[MOOSE and Lua Scripting 4](#_Toc44145748)

[MOOSE Documentation 4](#_Toc44145749)

[Essential Scripting Principles 5](#_Toc44145750)

[Lua Essentials 5](#_Toc44145751)

[Anatomy of a MOOSE Method 7](#_Toc44145752)

[Adapting An Existing Script 8](#_Toc44145753)

[MOOSE Scripting 101 11](#_Toc44145754)

[Lua Scripting Environments 11](#_Toc44145755)

[The MOOSE Framework 12](#_Toc44145756)

[The SSE 13](#_Toc44145757)

[Bookmarks 13](#_Toc44145758)

[Learning to Script Lua (for DCS) 14](#_Toc44145759)

[Running Scripts 15](#_Toc44145760)

[Testing A Script 15](#_Toc44145761)

[Principal of Inheritance 16](#_Toc44145762)

[A Little More Lua 17](#_Toc44145763)

[SPAWN 18](#_Toc44145764)

[Next steps 20](#_Toc44145765)

[How DCS Works 21](#_Toc44145766)

[Evolution of DCS SSE 21](#_Toc44145767)

[Script Execution Scope 22](#_Toc44145768)

[Anatomy of The Mission (.miz) File 22](#_Toc44145769)

[The Mission.miz\Mission Table 24](#_Toc44145770)

[Limitations 26](#_Toc44145771)

[Myths and Legends 27](#_Toc44145772)

[MOOSE Scripting 201 27](#_Toc44145773)

[Wrapper.GROUP 27](#_Toc44145774)

[Core.MESSAGE 28](#_Toc44145775)

[Core.SET 28](#_Toc44145776)

[Core.Zones 29](#_Toc44145777)

[Core.SCHEDULER 29](#_Toc44145778)

[Progress Recap 30](#_Toc44145779)

[MOOSE Scripting 301 30](#_Toc44145780)

[Core.EVENT 31](#_Toc44145781)

[Core.Coordinate 32](#_Toc44145782)

[Wrapper.Controllable 32](#_Toc44145783)

[Ready Yet For Assistance? 32](#_Toc44145784)

[Mission Environment Functional Classes 32](#_Toc44145785)

[Final Summary 33](#_Toc44145786)

[Advanced Troubleshooting 34](#_Toc44145787)

[The DCS Log and Errors 35](#_Toc44145788)

[Create a Test Mission 36](#_Toc44145789)

[Debugging 36](#_Toc44145790)

[How To Ask For Help 36](#_Toc44145791)

[Appendix 40](#_Toc44145792)

[History of MOOSE 40](#_Toc44145793)

[Snippets 40](#_Toc44145794)

# Document Purpose and Aim

This guide aims to equip a Digital Combat Systems (DCS) mission creator with the tools and knowledge required to write MOOSE scripts for DCS missions. The guide is organized in a modular and hierarchical fashion. For those who have heard of MOOSE and scripting but have no idea what those things are, the Very Basics module answers those initial questions. Subsequent modules build on each other, introducing increasingly complex features that one can assimilate as their MOOSE competency improves.

# The Very Basics

## What is a Script?

A script is a kind of high-level computer program written in plain text, and employing an underlying script “language”, to communicate behaviors to a parent program. The script language has reserved words and symbols that the parent program translates into an action or behavior. For our purpose the script language is Lua and the parent program is DCS.

## What is MOOSE?

Digital Combat Systems (DCS) comes with the Eagle Dynamics (ED) Application Programming Interface (API), also known as the **Simulator Scripting Engine** or SSE. **M**ission **O**bject **O**riented **S**cripting **E**nvironment, **MOOSE**, is a scripting framework that serves as a user-friendly interface for the SSE and is the brain-child of an extremely talented programmer named FlightControl. If you want to know more about this topic, check out FC’s “for dummies” videos found here:

<https://youtu.be/ZqvdUFhKX4o?t=618>

We recommend video playback at 1.5x speed, as FC speaks slowly and distinctly.

## What Can MOOSE Do For Me?

MOOSE can very efficiently implement mission actions difficult, if not (nearly) impossible, to do with the DCS Mission Editor (ME) trigger feature alone, like the following few examples:

1. Create a trigger zone of any shape.
2. Spawn random groups from random locations at random times
3. Continually spawn groups to maintain a desired number of units in the mission.
4. Trigger complex actions on occurrence of mission events (hits, deaths, births, spawns, etc.).
5. Create smart, responsive integrated air defense systems (IADS).
6. Modify group tasking based on mission conditions.
7. Use AI to direct player actions.
8. Manipulate placement and get distances between objects in the 3D world.
9. Get characteristics and manipulate large groups of objects.
10. Spawn and remove statics.
11. Easily create nested F10 menus.
12. Add mission environment features like an LSO or comprehensive IADS.
13. And so much more.

## What If I Don’t Want To Learn Scripting?

Fear not. You can still take advantage of many of MOOSE’s features by customizing one of the hundreds of scripts created by others and adding them to your missions.

## How Do I Get and Install MOOSE?

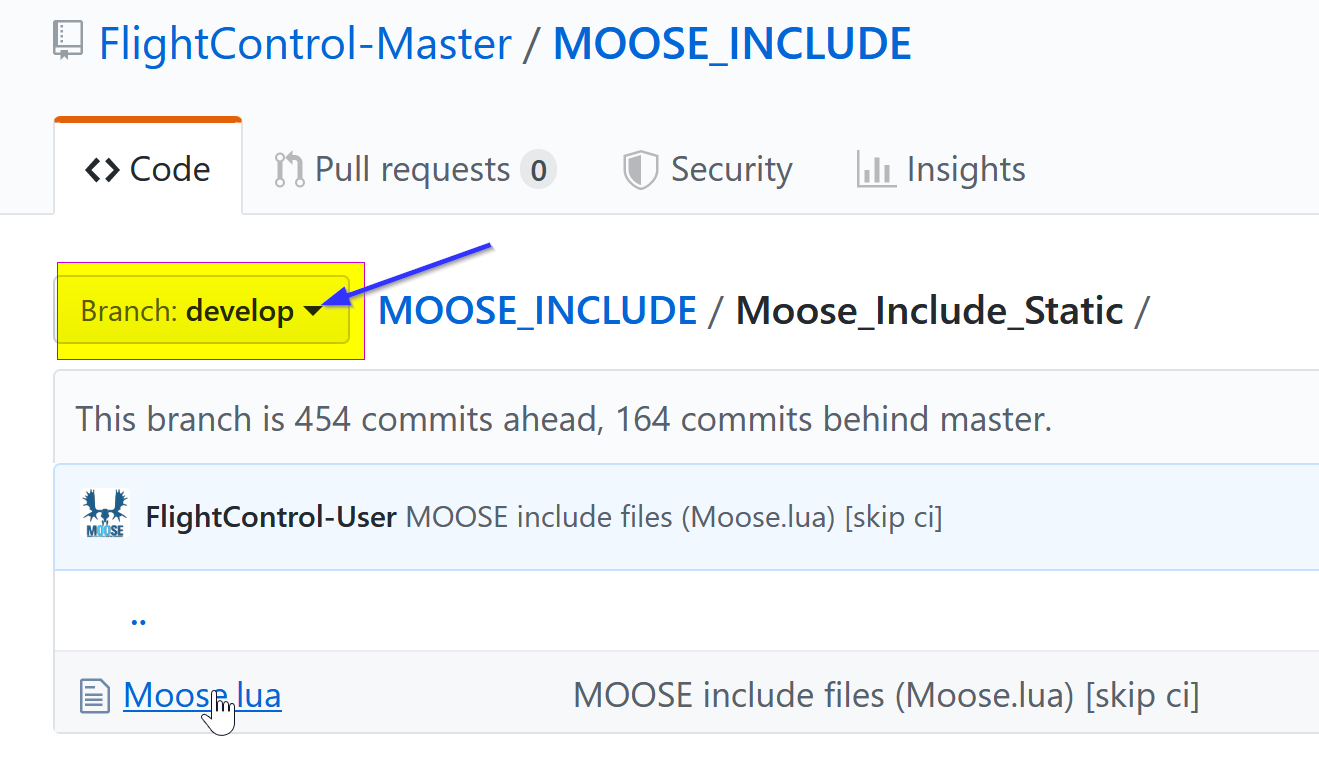
This section describes a quick and easy method of MOOSE installation aimed at those not interested in writing their own scripts. The main MOOSE file comes in two flavors, either of which works equally well: Moose.lua, the whole thing, and Moose\_.lua, a smaller-sized stripped-down version without documentation. The full public release version resides here:

<https://github.com/FlightControl-Master/MOOSE>

Get this version by clicking on the green “Clone or Download” button. A zip file containing all the displayed folders will download. Unzip the file and move it to a folder location of your choice. Use the Moose.lua in the Moose\_Include\_Static folder. The other downloaded stuff can be deleted if one does not intend to dive deeper into MOOSE script creation.

Get the most recent “beta” MOOSE version from the Github Develop Branch here, but be advised this version may not be bug-free. USE AT YOUR OWN RISK:

https://github.com/FlightControl-Master/MOOSE\_INCLUDE/tree/develop/Moose\_Include\_Static



To download the Develop version:

* Click on Moose.lua. Invokes a new window
* Click on Download button at lower-right. Moose script text will display.
* Right-click anywhere on the text. Invokes a selection list.
* Click on “Save Page As”. Invokes a Save window.
* Select desired folder, Click Save.

## How Do I Install MOOSE Scripts?

Moose.lua (and Moose\_.lua) file contains the instructions for running a MOOSE script and it must be loaded into the mission before any script file that uses one of its functions. Insert it in a mission using the DCS mission editor (ME) trigger action **DO SCRIPT FILE** at MISSION START. Then create a ONCE trigger to insert a dedicated mission script for your actual mission instructions, using a trigger action DO SCRIPT FILE or DO SCRIPT. Example:

**TRIGGER CONDITIONS ACTION**

MISSION START DO SCRIPT FILE (Moose.lua)

ONCE DO SCRIPT FILE (YourScript.lua)

Confirm the script has been added by opening the .miz file with a zip utility and observe the script files in the |10n\DEFAULT folder. Any reasonable number of different scripts can be included in the mission and all will run independently, barring conflicting variables.

## What Next?

Decide how deeply you wish to become involved with MOOSE scripting and start reading the relevant follow-on guide.

***Welcome to MOOSE and good luck!***

# 

# Using Existing MOOSE Script Modules

## Where Do I Find MOOSE Scripts?

Download scores of example missions and their associated MOOSE scripts from:

<https://github.com/FlightControl-Master/MOOSE_MISSIONS>

Sometimes short MOOSE scripts are called snippets. We have included a few in the Appendix and the MOOSE documentation includes many of them. We plan to create a repository of useful snippets at some point.

DCS introduces new program code at each patch, occasionally breaking a MOOSE function. Please report any non-functioning official MOOSE example scripts or missions here:

<https://github.com/FlightControl-Master/MOOSE/issues>

## How Do I Customize a Script?

MOOSE scripts can be read and modified using any text editor, so long as the editor saves the script in its raw form (Microsoft Notepad does **NOT** qualify). Notepad++ is one example of a freely available text editor that can be employed to create MOOSE scripts. Be aware that although complex MOOSE scripts can be created using nothing more than a common text editor, special Lua editor programs exist to help avoid script errors, thus saving time troubleshooting and debugging complex scripts. The *Mission Scripting 101* chapter of this guide discusses these more advanced script environment tools.

## MOOSE and Lua Scripting

Lua is a common scripting language and many tutorials for its use exist free on the internet, including the *Lua Online Manual:* <http://www.lua.org/pil/contents.html#P1>

Both the SSE and MOOSE are based on the Lua 5.1 scripting language. Lua employs most of the common protocols familiar to anyone experienced in program coding and is relatively easy to learn. Because MOOSE and the SSE are both Lua-based, raw Lua code, SSE (user flags, for instance) and MOOSE methods can all be used freely within the same script. The main Moose.lua file contains the code that serves as the interface between MOOSE scripts and the DCS SSE.

## MOOSE Documentation

MOOSE commands, called “methods”, are organized into categories, or “classes”. MOOSE documentation describes in detail how to employ the MOOSE methods; so when confused, refer first to the docs. Find them online here:

<https://flightcontrol-master.github.io/MOOSE_DOCS/Documentation/index.html>

Download the MOOSE documentation from here:

https://github.com/FlightControl-Master/MOOSE\_DOCS/tree/master/Documentation

## Essential Scripting Principles

To adapt an existing script to a mission requires understanding a few rudimentary scripting principles.

### White Space.

Lua ignores white space between lines and symbols, as well as at the beginning and end of lines.

### Comments.

Good script writing practice includes liberally adding non-executing comments to help others understand (and remind yourself) the intended behaviors of various sections of the script. A double hyphen “--” identifies a single line comment and Lua ignores anything following this symbol. Enclose multi-line comments with double hyphen, double square brackets, like so:

--[[ Many lines of comments --]].

### Define Every Word.

PC’s are basically stupid but they do work fast and never forget. This means that every word in a script must be defined or DCS will have no idea what to do with it. Now, fortunately a great many words and symbols are pre-defined for the script writer by the Lua language (if, then, +, -, (), true, false, math.random), as well as by the SSE and the main Moose.lua (SPAWN, MESSAGE, EVENT, GetCoordinate); but a newly introduced word/variable (RedCAP, for instance) must be defined. Lua features case-sensitivity: SPAWN does not equal Spawn.

### Define Every Object.

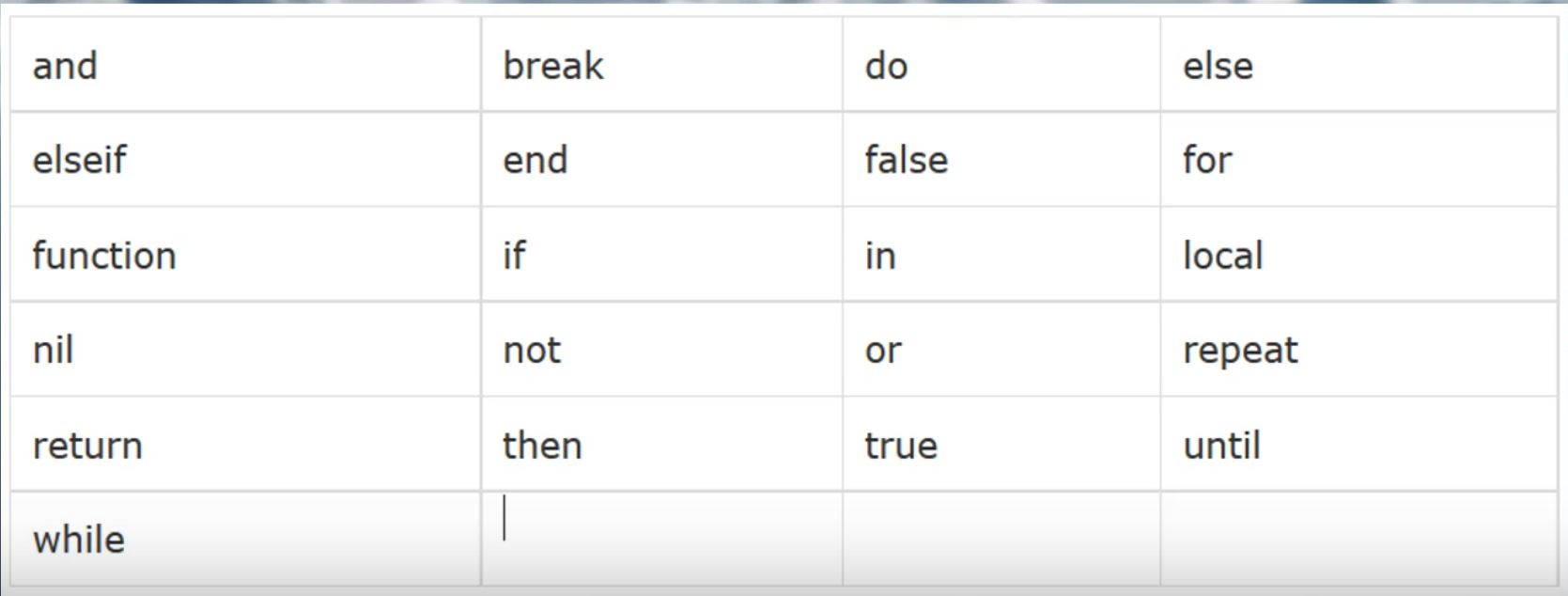
The script has no inherent knowledge of its intended mission; therefore, the script writer must define (“instantiate”) in the script every mission object mentioned.

## Lua Essentials

This section describes a few of Lua’s bare essentials. More detail on these concepts can be found in Chapters 1-4 of the *Lua Online Manual:*

### Variables.

Variables are named buckets of data created by the scripter, much like words defined in a dictionary. The names can consist of any combination of letters in the English alphabet and underscores. They are case sensitive and the only prohibited characters are numbers, special characters ( ( ) . % + - \* ? [] {} ^ $, etc.), apart from “\_underscore”, and these reserved words:



The key word “local”, identifies a “local variable”. Unlike global variables, local variables have their scope limited to the block where they are declared. A block is the body of a control structure, the body of a function, or a chunk (the file or string with the code where the variable is declared). Using local variables helps avoid the potential conflict of two different blocks of code containing the same variable name for different purposes. In such case Lua will regard the last identically named variable and ignore all others.

### Functions.

A function is basically a bucket also, but can be much larger than a variable bucket, and usually consists of multiple lines of script code. Functions are what make programs output ‘something’ (a ‘result’ in programmer parlance). The Lua language, the SSE and MOOSE all contain some pre-defined functions. A function in Lua doesn’t always need its own name but is always followed by brackets, even if there is nothing in the brackets, e.g. WriteFile(). Functions start with the keyword “function” and end with the key word “end”; and they can frequently become long and complex.

A function can (and should) return a result value after being called. Calling a function by writing its name produces the result directly. Function results can just return true or false, a number from a calculation or a variable. Functions can be reused, in that you can make your entire script a function, and then call it twice with two lines, rather than by twice copy-and-pasting the entire script.

### Arguments.

Arguments are the stuff inside a function’s brackets (), and they simply supply the data, if any, required by the function to produce a legitimate, precise result. Functions can have multiple arguments, each separated by a comma. A function’s arguments make it “replayable”, because a function called twice with two different arguments will return two different results.

### Booleans.

True and false <https://www.lua.org/pil/2.2.html>

### Strings.

Lua strings are a data type consisting of text in double quotes, for example: “I have a $30.00 bar tab”. Lua treats strings as literal constants or variables, accepting them, as written, for use in the script. <https://www.lua.org/pil/2.4.html>

### Relational Operators.

< > <= >= == ~= <https://www.lua.org/pil/3.2.html>

Note that novice scripters frequently confuse == and =. In Lua the equals sign on its own assigns a variable to a value, whereas the double equals sign tests if one value is identical to another.

### Statements.

if … then … else … elseif … end [https://www.lua.org/pil/4.3.1.htm](https://www.lua.org/pil/4.3.1.html)l

‘if’ statements are the bread and butter method for creating logic. They are best used in “either … or” type decisions.

### Logical operators.

And, or, not. <https://www.lua.org/pil/3.3.html>

Lua stops processing a statement when it returns true, so put the more important and common evaluations on the left side of statements.

### Script Processing.

DCS program Scripts processes scripts only once when triggered, and by line from top to bottom. The program stores variables and functions in memory for use when called by the script, and it will throw a fit if it encounters an object or variable not previously defined. Certain classes like EVENT and SCHEDULER are structured to repeat as the mission progresses; therefore, if a scripted action needs to occur sometime after initial script processing, that action must be initiated either by one of the special repeating class methods or by an ME trigger condition. Remember that any reasonable number of different scripts can be triggered to run by the ME trigger feature conditions.

### Units.

MOOSE default units are metric:

DISTANCE: meters

SPEED: m/s and occasionally Km/Hr.

### Altitude and Height.

In MOOSE method parameters, ALTITUDE is meters elevation above sea level and HEIGHT is meters elevation above ground level.

## Anatomy of a MOOSE Method

### Parts of a Typical MOOSE Method.

A typical MOOSE method, or function, looks like this:

SPAWN:SpawnInZone(Zone, RandomizeGroup, MinHeight, MaxHeight, SpawnIndex)

**SPAWN** in upper case represents, in this example, a previously defined object, named variable (a group in this case), recognizable by observing the next part.

**SpawnInZone** describes what the method does with the spawn group object. Guess what? This method spawns the object in a zone!

**Zone, RandomizeGroup, MinHeight, MaxHeight, SpawnIndex** all represent parameters, frequently optional, that customize the SpawnInZone. The MOOSE documentation for each method provides the parameters’ acceptable values. They must be in the form of a previously defined variable, one that defines the Zone for example, or a key word or symbol included in Lua’s native lexicon. RandomizeGroup, for instance, must be a boolean and MinHeight a numerical value (in meters).

**Example 1.** One common MOOSE method defines (“instantiates” in programmer’s parlance) in the script a group created in the DCS ME that is not late activated:

Variable = [**GROUP:FindByName( “Group Name” )**](https://flightcontrol-master.github.io/MOOSE_DOCS/Documentation/Group.html#%23(GROUP).FindByName)

**GROUP:FindByName** describes what the method does.

**Group Name** is the name of an activated group placed with the DCS ME. Any plain text in Lua (a string)must be enclosed in quote marks, like “Mustang”, otherwise Lua assumes the text is a variable and will throw a fit if it thinks it has encountered a variable that has not been previously defined.

**Example 2.** Some important methods take this form:

[SPAWN:New](https://flightcontrol-master.github.io/MOOSE_DOCS/Documentation/Spawn.html#%23(SPAWN).New)(SpawnTemplatePrefix)

which can used like this:

Variable = SPAWN:New( “Some ME Group” )

These methods with atypical syntax can usually be recognized by the word “New”, as well as by the fact that substituting a variable for SPAWN strips the method of any logical utility. What would Variable:New() do?? “Some ME Group” is the name of a group placed in the ME as late-activated and is termed a “template,” because the SPAWN method uses that group’s characteristics (number of units, type of vehicle, path etc.) to create a **copy of the template** in the mission.

## Adapting An Existing Script

So, finally, how do I adapt a script to include in my mission?

**Example 1.** A simple Group spawn script.

Spawn\_Vehicle\_1 = SPAWN:New( "Red Bandit 1" )

Spawn\_Group\_1 = Spawn\_Vehicle\_1:Spawn()

Line 1: Defines a variable (Spawn\_Vehicle\_1) to represent a new SPAWN object that is a copy of a late-activated group (the template) in the ME that is named “Red Bandit 1”. Note that this line only **DEFINES** the object for the script. It does **NOT** actually spawn the object.

Line 2: Assigns a variable (Spawn\_Group\_1) to represent an object in the mission, defined by Spawn\_Vehicle\_1, a copy of which is actually spawned by the function :Spawn().

The simplest adaptation of this script would change the line 1 spawn template name from “Red Bandit 1” to the actual group name in your custom mission in the ME. Note that in this script the spawn occurs immediately when the trigger DO SCRIPT FILE action conditions are met.

**Example 2.** This somewhat more complex example creates a polygon zone and sends a message when a designated group enters the zone.

-- Instantiate ME activated group Blue CAP. Substitute your ME group name.

GroupInside = GROUP:FindByName( "Blue CAP" )

--[[ Instantiate ME activated vehicle group Polygon Grp, whose route path defines the polygon zone. Substitute your ME group name --]]

GroupPolygon = GROUP:FindByName( "Polygon Grp" )

–[[ Instantiate a new polygon zone object in the mission with zone name Polygon A. GroupPolygon defines the zone.--]]

PolygonZone = ZONE\_POLYGON:New( "Polygon A", GroupPolygon )

-- Scheduler function, named Messager, periodically runs the included function()

Messager = SCHEDULER:New( nil,

function()

-- Condition statement checked at every run of the scheduler executes when true.

if GroupInside:IsCompletelyInZone( PolygonZone ) then

MESSAGE:New( “Polygon zone has been entered”, 15 ):ToAll() -- Sends message

-- Optional statement will stop the scheduler at first true condition

Messager:Stop()

end -- Ends “if then” statement

end -- Ends function()

--[[ Schedule parameters. {} == empty table. 0 == no delay from mission start, 1 == 1 sec. intervals. --]]

{}, 0, 1 )

**Example 3.** This script was developed to overcome a DCS bug of seemingly invulnerable AAA units.

-- mg1 is target group, consisting of 3 units, of interest in ME

MG\_Group = GROUP:FindByName( "mg1" )

-- Initialize MG dead counter, the number of units in the ME group mg1

MG\_Live = 3

-- Monitor weapon hit events on MG\_Group

MG\_Group:HandleEvent( EVENTS.Hit )

function MG\_Group:OnEventHit( EventData )

-- Identify unit hit

local Target = EventData.TgtUnit

-- increment counter on hit

MG\_Live = MG\_Live -1

-- Destroys target unit, when hit

Target:Destroy()

-- Return location of event target

Target\_Loc = Target:Getcoordinate()

-- Create explosion, intensity 75 (max == 1000) at target coordinate.

Target\_Loc:Explosion(75)

-- Send text message to players when MG\_Group suffers a hit

MESSAGE:New( "MG unit destroyed. Continue attack.", 15 ):ToAllIf( MG\_Live >= 1 )

-- When all of MG\_Group are destroyed, stop monitoring hits and send message

if MG\_Live < 1 then

MG\_Group:UnHandleEvent( EVENTS.Hit )

MESSAGE:New( "All MG units destroyed. Great job.", 15 ):ToAll()

end -- end of ‘if’

end -- end of function

# MOOSE Scripting 101

**So, now you want to write your own scripts.**

## Lua Scripting Environments

At this point you may wish to reconsider installing one of the dedicated Lua editor programs, or scripting environments, that provide “Intellisense”, auto-complete and other tools to assist the script writer to avoid common script errors. Eclipse Lua Development Tools (LDT) and Visual Studio Code (VSC) are two such programs.

### Lua Development Tools.

Lua Development Tools (LDT) is a Lua-specific text editor that detects and announces syntax errors in real-time. Additionally, LDT has autocomplete and “Intellisense”, as long as it is configured correctly. Intellisense automatically produces a text box displaying the documentation for a MOOSE method typed in a properly installed LDT Eclipse editor. Intellisense requires that the entire MOOSE project from Github be imported into LDT’s Eclipse editor. Follow scrupulously the procedure described in this YouTube video to setup LDT with Moose:

<https://www.youtube.com/watch?time_continue=1&v=C5yKu1BGpVQ&feature=emb_logo>

The video doesn’t cover everything, especially how to recover when things go wrong or when LDT needs repair after installation. Be aware that LDT forces a single specific workflow that may not suit a mature developer, but less-accomplished coders may find LDT a helpful script-writing tool. FlightControl produced a 21-minute video on the differences between certain text editors and the advantages of LDT here:

<https://youtu.be/fEbQDbxNGb8>

This video likely provides a lot of information superfluous to a novice’s needs, but it is worth watching, as it covers most of the available options. Download Eclipse Lua Development Tools from here:

<https://www.eclipse.org/ldt/>

### Visual Studio Code.

VSC needs two plug-ins, EmmyLua and IntelliJ IDEA, to have working intellisense.

VSC download and installation guide: <https://github.com/Microsoft/vscode/>

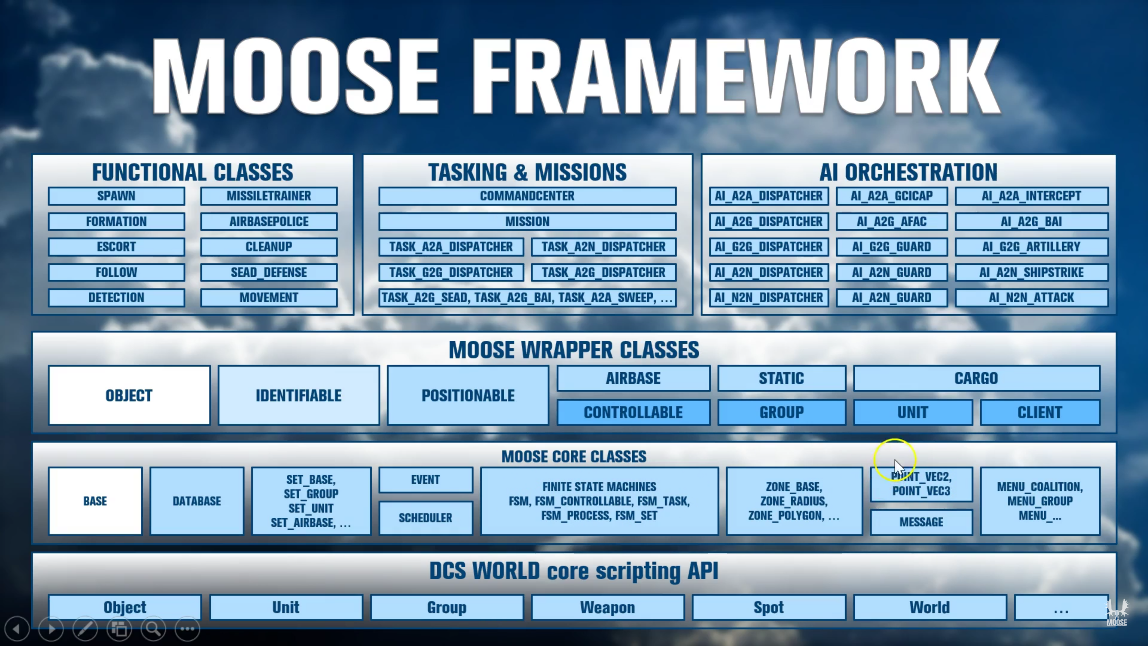
EmmyLua download: <https://github.com/EmmyLua/IntelliJ-EmmyLua>

InteliJ IDEA download: <https://www.jetbrains.com/idea/>

Bottom line: scripts must be written on some text editor and much better tools than Windows Notepad do exist.

## The MOOSE Framework

**M**ission **O**bject **O**riented **S**cripting **E**nvironment, MOOSE, is a scripting framework written in the Object Oriented Programming method. MOOSE derives all its magic from the Eagle Dynamics Application Programming Interface (API), also known as the **Simulator Scripting Engine** or SSE, an inherent feature of DCS. It allows MOOSE to do some cool things, ... but not everything. MOOSE sits on top of the ED scripting functions to provide the script writer with a more powerful and understandable bridge to the SSE. Moreover, MOOSE extends and combines several SSE functions into entire classes of derived features. MOOSE’s lowest level is the Core classes, which sit just on the SSE and suck in the SSE’s information to a DATABASE.



Other base classes create the Schedulers, Finite State Machines and tools that MOOSE grows from.

Next up, MOOSE wraps the existing SSE classes like GROUP and handles their registration into its own database for further use. After that comes higher level functional classes that actually do things, but also some foundation code layers like Detection, Spawn and Task, which build into entire AI dispatchers that detect, spawn and task AI entities. The functional level also has standalone mission environment classes like Random Air Traffic (RAT), AirBoss, Missile Trainer, etc. that can live on their own at the high-level Functional Class area. This is all explained in FlightControl’s initial “for dummies” videos found here:

<https://youtube/ZqvdUFhKX4o?t=618> (recommend 1.5x speed).

## The SSE

While the ED Application Programming Interface (API), or Simulator Scripting Engine (SSE), enables MOOSE to do some cool things, MOOSE has no capabilities beyond those enabled by the existing API methods. The types of things possible include creating new units mid-mission, using timers, using the Lua framework to access the objects in the mission, creating sounds and messaging, fine control of units etc, tasking, detecting, patrolling and managing via clever use of programming, like **Finite State Machines** (FSM).

What ED’s API does not allow is access to manipulate clients or their aircraft in multiplayer. The API restricts available information and control, and does not allow changing AI behaviour itself, only to initiate pre-scripted behaviors. MOOSE uses and organizes these API’s to make them more accessible to the script writer. Scripts that might take ten lines of raw API become two in MOOSE. MOOSE takes an API function like “coalition.addGroup()” and makes it into a MOOSE function called “SPAWN:New()”. SPAWN typically requires one line of script, whereas coalition.addGroup consists of a very large table of groups and units and other things, totalling over 50 lines or more. Moreover, MOOSE creates entire modules, such as **A2A\_Dispatcher** that greatly simplify processes that would prove daunting using raw API methods.

## Bookmarks

Bookmarking from a browser may seem archaic for the children of Google accustomed to navigating the internet using browser history. Mr. Google, however, remembers all the naughty children and where they have been on the internet, so clear your internet history and use bookmarks!

### MOOSE Bookmarks.

These browser bookmarks will prove very useful resources during your MOOSE scripting journey.

Landing page for documentation (the full version of MOOSE.lua also contains all the documentation):

<https://flightcontrol-master.github.io/MOOSE_DOCS_DEVELOP/Documentation/index.html>

MOOSE release version.

https://github.com/FlightControl-Master/MOOSE/releases

MOOSE development version static file.

<https://github.com/FlightControl-Master/MOOSE_INCLUDE/tree/develop/MOOSE_Include_Static>

The main repository of all the demo missions that illustrate the utility of a great many MOOSE functions.

<https://github.com/FlightControl-Master/MOOSE_MISSIONS>

The main page for MOOSE, where one can log issues, review MOOSE issues reported, make pull requests, view the code changes and all that good stuff.

https://github.com/FlightControl-Master/MOOSE

### FlightControl’s YouTube page.

https://www.youtube.com/channel/UCjrA9j5LQoWsG4SpS8i79Qg/videos

### Hoggit Bookmarks.

Grimes, who has long been a great help to DCS scripters, keeps these three resources current, mostly Hoggit Wiki pages. The functions are super important, but these pages hold more details like enumerators and weapon types and categories that every MOOSE script writer eventually needs.

https://wiki.hoggitworld.com/view/Simulator\_Scripting\_Engine\_Documentation

https://wiki.hoggitworld.com/view/Category:Singleton\_Functions

https://wiki.hoggitworld.com/view/Category:Class\_Functions

### Lua-Specific Bookmarks.

At some point all Lua script writers will encounter issues requiring the information held by these two references:

[http://Lua-users.org/wiki/TutorialDirectory](http://lua-users.org/wiki/TutorialDirectory)

[http://www.lua.org/manual/5.1/](http://www.lua.org/manual/5.2/)

Stackoverflow.com and the ED Forums hold much valuable information for Lua scripters, as well.

## Learning to Script Lua (for DCS)

Folks learn things by various means. Some are kinaesthetic learners (monkey see, monkey do). Some are visual learners and can handle abstracting words, while some are visual and need pictures, or video. Some can interpret aural instruction better. Actually, however, **we learn through a combination of all of the above**. Unfortunately for most of us, true comprehension proves elusive. Copying existing script material is a valid initial learning exercise, but without comprehension, one must rely on a supply of new material and cannot conceive custom code to achieve a novel mission behavior. As a goal, strive to acquire an understanding of basic Lua and the skill to create new MOOSE scripts, although code cut-and-pasting will inevitably remain an important script development tool.

Mastering fundamental principles of any general topic facilitates faster mastery of its derivative elements. This is why we have ‘101’ courses and why learning proceeds in a sequence that builds upon previous knowledge. This building-block concept is why children learn addition before multiplication, and multiplication before division; and why missing a critical math lesson can render a child permanently deficient in math skills.

This section provides some common advice, pitfalls and valuable resources. This guide provides the tools, but the new scripter must employ them productively. To proceed successfully, the new script writer must have:

* Learned the makeup of a mission file and the DCS mission environment and table
* Setup Eclipse LDT or a script writing tool
* Found a log tailer to track logs in real time
* Discovered the location of the MOOSE.lua (dev and released versions)
* Become familiar with researching the MOOSE documentation.
* Bookmarked the Hoggitworld Wiki API functions
* Learned how to execute a script dynamically with ‘loadfile’
* Understood how to interpret log errors and ask for help

## Running Scripts

The DCS ME has four locations where Lua script can be directly introduced without using a text tool.

* In the Mission Editor triggers, as a DO SCRIPT - a small text box appears for directly writing or pasting code.
* Lua can also be directly written into a conditional mission trigger of “Lua PREDICATE”. It requires a Lua statement to be either true or false (Boolean) response.
* Another Lua Predicate exists at an advanced waypoint action where a DO SCRIPT can be written (or pasted).
* Selected advanced tasks like ON LANDING also have text boxes for scripted conditions.

FlightControl’s video explains this in detail here:

<https://youtu.be/fIp2VdvRXQw?t=79> (recommend 1.5x speed)

## Testing A Script

Rare indeed is the script that functions flawlessly at the first trial. The mission creator can easily find him/herself altering and re-testing a script dozens of times before arriving at the Eureka! moment; therefore, we recommend employing a procedure more efficient than simple DO SCRIPT FILE script loading during mission development phases. Additionally, when using the simple DO SCRIPT FILE procedure, the compiled script is placed in the Windows Temp file path under a DCS folder and given a temporary name, complicating locating code error script line numbers in the DCS logs. Consider, therefore, adopting the following procedure to improve script troubleshooting efficiency:

1. Open the DCS ME in a window.
2. Use Lua’s loadfile() function to run the script directly from your hard disk.

assert(loadfile("D:\\A\_Folder\\Another\_Folder\\ScriptFile.lua"))()

Note the double backslashes \\

This line in a DO SCRIPT trigger action text box calls the script directly from its hard drive location. Then, the scripter can modify his work in the script editor and run the mission directly from the ME without restarting DCS and without re-saving the mission. LSHIFT+R is the most useful script editing key bind in DCS, as it restarts the mission. This procedure also implements printing the script file name (if you use multiple scripts) in the DCS.log, as well as printing the script line numbers of log entries, to facilitate locating script errors.

As another option, one can also launch the mission with the revised script without restarting the mission, by placing the DO SCRIPT as an action called by a Menu Item. Make the DO SCRIPT CONTINUOUS, by creating a user flag that *resets* to a constant number on every activation of the F10 menu item. Then the script can be run multiple times until it performs correctly without restarting the mission or returning to the Mission Editor. This procedure is explained in detail in this video:

<https://youtube/BMKBXjjKiDI>

Rest assured that these efficiencies will save much time better devoted to bar-hopping and bring mind-blowing clarity to error messages.

## Principal of Inheritance

By virtue of MOOSE’s hierarchical nature, certain classes “inherit” methods from a “parent” class. The class’s parent, if any, is stated in the MOOSE documentation; and the script writer has available all the methods from the parent class, even though the child class documentation may not restate the inherited methods. For example, the GROUP wrapper class is a child of the CONTROLLABLE class, which is a child of the POSITIONABLE class, which is a child of the IDENTIFIABLE class, which is a child of the OBJECT class. The IDENTIFIABLE class, therefore, includes all the methods documented in the OBJECT class. The POSITIONABLE class includes all the methods documented in the IDENTIFIABLE class, as well as its inherited methods, and so on. The GROUP wrapper class, then, includes all the methods in its family tree.

## A Little More Lua

### Tables.

A tutorial on Lua tables can be found here:

<http://lua-users.org/wiki/TablesTutorial>

Tables will start easy and get hard. They are used frequently to store lists of group names for random selection in MOOSE. Be aware that the Lua language has some unique/unusual rules for tables. One will eventually need to understand indexing and Lua’s automatic indexing system. What is an Index? An index is a little tag that organises table items so they can be found faster, bit like a large ring binder with coloured tabs sticking out to mark interesting parts. The most common and practical index method is numerical (1, 2, 3, 4, 5 etc.). Here is a trivial Lua table example:

Table = {‘thing1’,’thing2’}

Lua automatically assigns **thing1** the index of 1 since it’s the first item in the list. When no index is specified, Lua decides its own Indexing, starting with [1] (not zero like other languages); so for

Chest={“socks”, “pants”, “Shirts”, “trousers”}

then Chest[3] is “Shirts”. One can define the indexes manually for other reasons:

Table = {[1]=’thing1’}

In that case, [1] is the index for ‘thing1’. Other terms for index are “key, value pair” or associative arrays. In Lua, the key can be another string like

Chest = {[“First Drawer”] = ‘socks’}

Pull the socks out of the drawer with this:

Chest[“First Drawer”]

Returns ‘socks’.

Note that tables **do not require** indexes. In such case Lua orders the table’s items haphazardly. See more on pairs and ipairs. Note also that Lua ignores a trailing comma after the last table entry:

{“thing”,} -- extra comma doesn’t mess it up

### Concatenation.

Concatenation is joining two things together, usually to output them as a single line of text, where part of the text is evaluated code. Concatenation is a double fullstop/period -- not one, not three, but two. Four is way off. Concatenation plays the very important role of attaching strings of words to functions that turn into words to read a complete sentence in English.

A = “Mickey Mouse”

MESSAGE:New( “My tank is called “.. A, 15 ):ToAll()

Outputs an in-game message: “My tank is called Mickey Mouse”.

### For Loops.

‘For loops’ access a table item when its table location, or its existence, is unknown. The ‘for loop’ iterates the table (sequentially selects each table item) to test each item against the ‘for loop’s’ conditions. For using for with key, value in pairs and ipairs see:

<https://www.lua.org/pil/4.3.4.html>

<https://www.lua.org/pil/4.3.5.html>

Example:

Chest = { “Socks”, “Underwear”, “Shirts”, “Trousers” }

for i = 1, #(Chest) do

if i = “Socks” then

Chest[i] = “Ties”

end

end

This says, in English**:** The variable “i”, represents a number from one to the number of entrys in the table ‘Chest’. Do count from 1 to that total entry number, and check if Chest[i] == “socks” for each index, then replace Chest[i]’s value with the string “ties”. After executing, the table Chest should look like,

{“ties”, “underwear”, “shirts”, “trousers”}

The code did this: Is “socks” == “socks”? Yes, ok replace that entry with “ties”.

‘For loops’ are more commonly used instead of ‘while loops’ in DCS, as ‘while loops’ can become an infinite loop that would crash DCS. “While 1” loops are not used in the main programming block of DCS.

This concludes our treatment of the basic anatomy of Lua. Lua has many more features, but this presentation covers 99% of DCS scripting possibilities.

## SPAWN

Experience tells us the best introduction for learning MOOSE is seeing how the most common MOOSE function works and getting some instant gratification. So, let’s jump right in with Spawn.

All things appear from SPAWN (out of nowhere, apparently). We present this class first, because it has proved the most useful, powerful, easy and impactful function on the scripting learning path. Literally everything dynamic and interesting starts from the ‘rabbit out of the hat’, i.e. SPAWN. This is especially true for mission designers who are sick of flying their own static missions. Head to the SPAWN docs and have a read to become familiar with the class’s capabilities. Next grab a demo mission and experiment to see what it does.

### Analyzing SPAWN.

Now try to follow and understand the MOOSE file that corresponds to the SPA-14**,** SPAWN demo mission:

Spawn\_Vehicle\_1 = SPAWN:New( "Spawn Vehicle 1" ):InitLimit( 5, 0 ):SpawnScheduled( 5, .5 )

The wording implies something is being created; however, having understood the previous section on Lua’s basics we should look at this in a structural way first to understand it better:

Variable assignment = CLASSFunction:Function(arguments):Function(arguments):Function(arguments)

The ‘Something’ of our previous examination of SPAWN has become “:Do something (this way) :Do something (this way) :Do something (this way)”.

* An Assignment gives the variable (the bucket) contents (a value).
* Arguments provide a mechanism, sometimes optional, to customize the function’s execution.

In this case, the variable Spawn\_Vehicle\_1 holds all the information returned by the running of these three following functions. The variable does nothing on its own, and is optional in this case because SpawnScheduled does the actual spawning, but is useful for later reference. In this case Spawn\_Vehicle\_1 holds the ‘Object’ that is the Spawn instructions used to create a copy of the ME template “Spawn Vehicle 1”.

The three functions execute according to their arguments, and the SPAWN documentation describes how the arguments of the functions of New(), InitLimit() and SpawnScheduled() control the result. For New(), it’s saying, use the late-activated Group called “Spawn Vehicle 1”. For InitLimit(), it’s saying ‘limit the spawning to 5 units and unlimited groups’; and for SpawnScheduled() it’s saying ‘spawn every 5 seconds and vary the spawn time by up to half either way of the original 5 seconds’ (i.e. 2.5 - 7.5 seconds).

Do note that the last function, SpawnScheduled() is the function that actually places the template copy in the game world. Without SpawnScheduled(), all the defining characteristics of the SPAWN have been created, but the object itself has not been commanded to spawn. This peculiarity of SPAWN provides reason enough to read the documentation.

### Another Way to Do It.

Now, to demonstrate how Lua does not always require a single rigid construct, this totally different code invokes an identical mission result.

MySpawn = SPAWN

:New( "Spawn Vehicle 1" )

:InitLimit( 5, 0 )

MySpawn:SpawnScheduled( 5, .5 )

The bucket (variable) is renamed MySpawn. The method segments appear in multiple lines (remember Lua ignores white space); and, finally, SpawnScheduled() directly spawns a copy of the ME template in DCS from the information in the variable MySpawn. Observe that the variable MySpawn was assigned to the MOOSE object defined by the characteristics of New() and InitLimit().

### But What Does the Colon ( : ) Do?

Few scripters express curiosity about the lowly, but essential, colon; but its properties should be understood. In Object Oriented programming the colon enables the “chaining” of functions. Specifically, by using a colon operator, the name of the object need not be passed as a first argument:

myobj:foo(n) == myobj.foo(myobj, n)

In other words, the colon serves to automatically pass the Object into the next function, thus enabling “:function():function()” type of writing, a common occurrence in MOOSE.

Absorbing all this information resembles drinking from a firehose; but no easy way exists to explain programming basics. The next steps will involve some trial and lots of error.

## Next steps

Stop using the demo missions.miz and start experimenting with MOOSE functions. Run the SPAWN demos. Experiment with them. Work with unfamiliar functions and Lua statements and see where they break. This stage of learning MOOSE focuses on understanding MOOSE’s capabilities, its potential. A MOOSE scripter must know the functions and what they produce. It’s a chicken and egg scenario. How can one begin to create novel mission activity without knowing the MOOSE tools available and their limitations?

This is an iterative cycle. The scripter discovers a few functions that inspire an idea. Then s/he researches the MOOSE resources to determine how the idea might be realized in a mission and attempts to achieve it. If successful, great! If not, try to achieve some similar activity and so on. Do not hesitate to try new things, as broken missions are infinitely repairable.

Allen Saunders once said, “Life is what happens to you while you are busy making other plans”; and learning to code is quite similar. That is, learning occurs whilst making code mistakes and corrections, especially if the error was exasperating and time-consuming. Watching videos and reading books as a learning experience, though useful, pale in comparison.

Expect to experiment with SPAWN in one- or two-line scenarios for several days, discovering its power and its limitations. SPAWN’s more advanced features include:

* OnSpawnGroup()
* Randomization
* Multiple spawns from an array in a “for loop”
* InitLimit() can fail because one neglected to read the arguments or doesn’t know that UNIT and GROUP classes have different functions.
* Spawn() returns the group name
* Using ZONEs and tables

Be aware that while experimenting with SPAWN, one may well encounter shortfalls in the documentation for InitCleanup(), try it (incorrectly) on ground groups and discover that using InitCleanup() inside air bases, as opposed to outside air bases, yields different results.

# How DCS Works

## Evolution of DCS SSE

Two types of users inhabit the DCS world: those who care nothing about DCS’s inner workings, and those with an interest, but no knowledge, regarding DCS and how it works. Surprisingly ED has only two, yes two, ‘testers’ with any capability in Lua and they have a lot on their plates. From outside the ED team discerning the guts of DCS constitutes black art, undocumented and obscure with no aids.

The simulator has an archaic, creaky engine. Its original designers have become largely unavailable and studying the engine leads to black holes and frustration. Shocking as it may seem to millenials, who rely heavily on YouTube videos and Google search, experienced people remain gold mines of information. Those outside the ED team gained much of their established ‘knowledge’ of the engine through arduous trial and error requiring hundreds of hours of smashing faces against keyboards. As a further obstacle, DCS was written by a Russian parent company; and Russians, in general, have unique communications cultures. Moreover, the ED team must protect its valuable military connections while serving its primary DCS market, the western world.

DCS has multiple Lua environments, each totally insulated from the others and having its own distinct scope. The Lua environment of interest to MOOSE scripters is the “Mission” environment. The other key Lua environment is the “net” environment, which runs on the DCS server and has a different, documented API available in the DCS install under the folder “API”. These two environments are isolated from each other for security reasons.

Circa 2015, Ciribob found that both environments share global flags. This discovery led to such scripts as Simple Slot Block, which uses Mission environment information (groups and flags) and executes a net function in the net environment: *onPlayerTryChangeSlot()*.

Meanwhile, SLMOD by Speed and Grimes used a Lua TCP connection to bridge the two environments. “Web-comfortable” designers, Java and Python folks, developed interesting web applications relying on exports to the cloud. Within ED the DCS ‘mission’ guys sat isolated and confined, whilst the SSE accumulated bugs.

One CAN run MOOSE (or any Lua) in the Net environment in the Saved Games\DCS\Scripts\Hooks area by creating a GameGUI hook for the script. Any modifications, however, need a full restart of the game engine; so, working in the Net environment proves problematic. This protocol likely harbors numerous bugs and has never been fully tested. Workings of the Net environment lie beyond the scope of this document; but, if interested, look at how Tacview uses its hooks and slips lines of code in between frames of execution.

## Script Execution Scope

By default, scripts executed from the mission have no access to the local file system, IO or operating system (os, io, lfs). Moreover, scripts can only call functions from the DCS mission API.

<http://www.lua.org/manual/5.1/manual.html#5.7>

One can, however, increase the scope of Lua by stopping DCS from sanitizing these libraries in the Scripts\MissionScripting.lua file by commenting out these lines like so:

sanitizeModule('os')

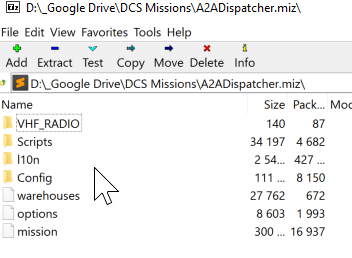
-- sanitizeModule('io')

-- sanitizeModule('lfs')

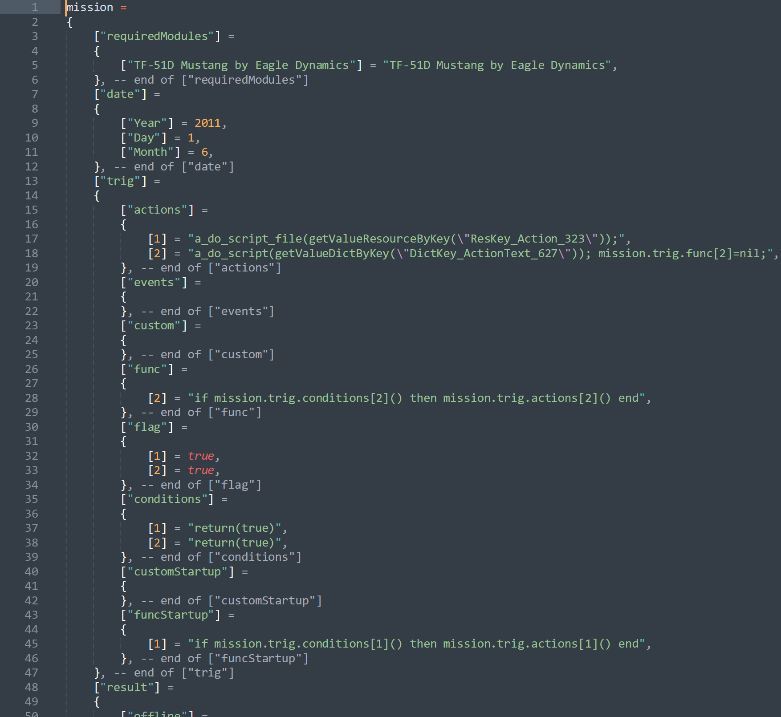
This procedure does not affect multiplayer server integrity checks (IC).

These edits do allow access to the file system and make available functions that can wreak havoc, if assisted by a careless PC user who downloads and executes someone’s mission without reading its scripts, a pretty dumb chain of events; but it does constitute a risk. Due diligence executing downloaded files all but eliminates this risk.

## Anatomy of The Mission (.miz) File

DCS scripts, running in the mission environment, launch into the ‘mission’ environment and stay isolated from the ‘net’ functions, unless circumvented by using global variables set by the userFlag functions. The “missionfile.miz” file is simply a .zip file readable with any utility that decompresses the .zip file format. Its data creates the mission environment and loads into memory at the start of the game execution. Understanding the mission.miz file structure has many benefits for the script writer, as it is reused in scripting in several places and helps to understand the broader picture of DCS. The mission file, or “.miz”, anatomy somewhat resembles the DCS folder and the Saved Games folders. The following describes the .miz included folders, not all of which will be found in every mission.

**File; \mission.** This is a large, complex table created by the mission editor and read at runtime. It loads the units and groups, time/day and coalitions, countries, weather, triggers and such into the game engine. DCS reads the mission file only once at mission runtime; therefore, it cannot be manipulated thereafter. Access its table output by asking for ‘env.mission’.

**File; \l10n\DEFAULT\dictionary.** This file helps localization (dev speak for language support). It translates names of things from a generic name to a local name in the DCS user country’s language. The file also holds trigger actions and name details.

**File; \warehouses**. This file lists all the weapons, units and fuel available in DCS Resource Manager. Each air base, farp, oil rig, ship with player landing services and static object warehouse is stored here. Whilst most designers set warehouse inventory to “Unlimited”, the DCS partially working and complex warehouse system enables resource transfers, like ammo, between warehouses during the mission. Warehouse system development appears to have been abandoned, at least temporarily, but our warehouse use guide can be downloaded from here:

<https://forums.eagle.ru/showthread.php?t=183207>

The most unread but most-owned book (not the Holy Bible), the *DCS User Manual* in the ‘Doc’ folder of your DCS installation, has a useful description of the warehouse system. We recommend periodic re-visiting the DCS User Manual, as every rereading seems to reveal something new and wonderful!

**Folder; \|10\DEFAULT.** This folder stores mission scripts, among other things. Beware that the DCS ME manages this folder and can delete user files, sound files for instance, unreferenced in the mission.

**File; \Track and Track Data** folders populate from a saved track file, a recording of the mission run in Multiplayer. These files hold all the recorded pilot keypresses, cockpit switching and controller inputs. Mission events record sequentially and save into the mission.trk. A ‘.trk’ is merely a .miz with track data and plays only in the forward direction. Unfortunately, DCS records track data inaccurately and playback may not reflect actual mission events, unlike what Tacview has achieved.

**File; \Scripts** folder contains some legacy Lua files on weather and earth GPS. Illogically, mission scripts reside in \l10n\DEFAULT.

**File; \Config\View** folder contains View.lua, SnapViewsDefault.lua files, whichdo overwrite user settings. It also contains Server.lua with server-side controls on views, that will overwrite the default players’ settings.

**Cockpit Setup Folders.** Using “Prepare mission” from the Mission Editor will create, for now, KA50 and A-10C customised cockpit folders in the .miz file, saved as the device name, e.g. “ABRIS”. These configurations overwrite the player experience.

**User Folders.** The user can create custom folders in the mission file and fill them with any desired files. Custom folders added to the .miz zip will remain largely untouched by the mission editor. The ME checks these user folders for extraneous media files declared in the triggers and loads them into the mission. As previously stated, at mission save the ME removes sound files manually added to the .miz, and not referenced in a trigger action.

**File Precedence.** The two areas that store configuration settings are the \Saved Games\DCS\ folders and the DCS main installation directory. Settings in the DCS installation folder pass in a hierarchical fashion into the game environment. First comes the Game’s default settings, then the player configured changes from Saved Games\DCS, and finally, the mission zip (.miz), which overwrites all client settings. That is how a client’s settings and views get overwritten when joining a server.

## The Mission.miz\Mission Table

Generally speaking, the key to advance one’s script writing competency is understanding the format of the core underlying tables, which reveal such concepts such as Task, Mission, Routes, Group and Units, their attributes and references. If MOOSE scripting were a three-year university course, Tasks and Routes would be in year three.

The mission.miz\mission file, is a huge table containing the instruction set for building the game environment. Its structure can prove confusing, but the mission file table plays such an important role in DCS scripting that it merits its own section in this User Guide. This table contains the definitions for all the mission entity, country and coalition attributes; and a script can access this table during mission execution.

Any table in Lua can nest inside another table, and the mission table resembles a Russian doll of nesting. Everything from the start of the mission resides in this table and its most important element is the mission table’s GROUP structure. Why? Because any GROUP or UNIT data requested by a script resides in this portion of the table. Spawning a group using the SSE, without MOOSE or MiST, requires that the script exactly duplicate the mission file table’s arcane group structure. The table also provides clues regarding available group information obtainable during mission execution! The mission table follows this structure for each group:

Mission["coalition"]["blue"]["country"][28]["ship"]["group"]

The red or blue coalition comes first; then the country, of which there are many, enumerated by a number; then the group types of air, ground, ship etc. Under the ‘group’ comes its group number and all the group data from the ME. Getting a unit’s ‘x’ coordinate requires drilling down to:

Mission["coalition"]["blue"]["country"][28]["ship"]["group"][1]["units"][1]["x"]

That’s a long way!

The GROUP is a parent structure for units. It contains all the instructions for its units, for example the group’s location (not each unit location, that can blow your mind away), its waypoints, its tasks, as well as whether it is hidden on the map, late activated, its name, and other attributes. 

‘**Tasks’** (plural!), a high-level AI instruction, are a key, but confusing, element of the GROUP structure. In the example to the right, if a group has the primary mission task of “CAS” in the ME, CAS persists as an action at all waypoints, as if inherited across all waypoints. This ‘Tasks’ [sic.] embodies a single GROUP-based list of persistent behavioural instructions. CAS, in this case, consists of nothing more than “Engage Targets” with target types of Helicopters, Ground Units and Light armed ships.

The other type of instruction is the (Combo) **‘Task’** (singular!). This ‘Task’ contains a lower-level set of instructions, specific to an individual waypoint. Multiple Combo Tasks can be issued for each waypoint.

Below the GROUP reside its UNITs. A set of data for each UNIT provides the identification, type, location, heading, speed, start points, armaments, livery etc. The group controls its units. Send the group one way, all the units must follow; however, a group can consist of different types of units belonging to the group’s main category of ground, air or naval. Statics are groups with no unit inside them and have slightly different properties.

Clients are units of type ‘Client’. Players are the same, except they become the focus of a single player mission. A mission can have only one unit of type ‘Player’, into which the human operator will automatically spawn. On the other hand, a mission can have multiple Clients and a human can choose which client to spawn into. The mission file provides a great many clues on issuing group-level tasks and Combo Tasks.

## Limitations

Recall that DCS reads the mission.miz file only once, at runtime, and copies it to a Windows\temp directory. The DCS game engine continuously updates the data in the Windows\temp copied table as the mission progresses after runtime; but a script can not, during the mission, access the original mission.miz file. If, during the mission, a group receives a new order from, say, Combined Arms, the original .miz table orders cannot be recalled to discover the group’s primary tasking. One can obtain, however, the group’s current location, health and ammo status from the continuously updated Windows\temp table. The same goes for elements like client slots, as new slots cannot be created after runtime. Nor can one move warehouses. Warehouse current inventory can be altered, but its original stores and attributes cannot be accessed after runtime. Ships can circumvent this limitation due to their special coded properties.

Common API limitations script writers run into include:

* Although no teleport command exists, delete-and-spawn simulates teleporting.
* Client aircraft cannot instantiate with initial damage.
* A script cannot access Client cockpit arguments. Not so for Player arguments.
* Dead vehicles are removed from the mission and are inaccessible. Dead statics remain accessible, though. Go figure!
* Client aircraft cannot be moved or affected. Previously we could destroy them, but currently a bug disappears dead clients in view of all other clients!
* Weather and time of day are unchangeable.
* FARP’s and Client-based ships cannot be spawned
* AI’s tasks remain inaccessible.
* Client waypoints are fixed.
* Green training gates exist but are invisible in multiplayer.
* Player slots have access to certain functions that Client slots do not have; and the Player slot is designed for Single Player, where these functions operate in a different environment, e.g. gates and cockpit arguments.

## Myths and Legends

‘Flat world theorists’ commonly believe that scripts slow down mission execution, reduce FPS, cause diseases and infertility. In fact, a few lines of code run so fast that the 100th of seconds timer in the logs shows them running in the same instant! To achieve a degradation of DCS performance one must either employ some very clever programming, or blindly stumble upon a method that degrades DCS program performance.

First, because the mission reads a script only once, at runtime, the script cannot affect CPU performance subsequently. Second, a statement so elegant as to instigate such intense processing in a tiny loop, sufficient to kill DCS, could not possibly be created by a casual mistake. An example of a performance-killing script is one that calculates every coordinate of a MLRS battery rocket barrage, or cluster bomblets (when those were separate entities) and then performs some complex logic on the results.

The last reason scripting is mostly irrelevant to performance is that DCS itself is quite capable of melting CPU’s and creating a slideshow with no outside assistance. Exploding objects on-screen, for example, requires not only GPU processing and other explosive effects, but also latches onto an in-game ‘dead’ event and provides a set of nested ‘if’ statements requiring extensive lookups and logic.

# MOOSE Scripting 201

Experience with the DCS ME will already have familiarized a mission designer with GROUP and UNIT entities. MOOSE “wraps” these items and also PLAYER, CLIENT and CARGO entities and many other key parts of DCS, into accessible tables in the database. MOOSE scripts can never refer to a group or unit by its ME name, but rather to its wrapper object, which we instantiate in the script by declaring it.

## Wrapper.GROUP

The GROUP wrapper is one of the most interesting MOOSE classes because it reveals the great depth of information obtainable from certain game entities. Testing can reveal numerous group attributes, such as existence in a zone, being alive and being airborne, as well as its speed, type, number, absolute in-game position, and count the number of its units existing in a zone. One of MOOSE’s most confusing aspects for a new scripter involves finding the name of a group spawned dynamically. The SSE returns the group name to the script when a SPAWN is commanded, like this:

MySpawn = SPAWN:New(‘template’) -- Instantiates the Spawn Object, copying attributes of ‘template’

MyGroup = MySpawn:Spawn() -- When called, Spawn() returns Group object spawned

HisName = MyGroup:GetName() -- Knowing the Group Object you can use GROUP functions on it

MESSAGE:New(“My group’s name is “ .. HisName, 15):ToAll() -- prints “My group’s name is template”

Read the GROUP Wrapper class docs first to become familiar with the type of information available. Then experiment with SPAWN to discover its group properties and print them to screen. This is the optimum path to initial MOOSE scripting skill, but expect mastering the GROUP methods to present some challenges. Group property information derives from the group section of the mission table, as discussed previously in *The Mission.miz\Mission Table* section.

## Core.MESSAGE

MESSAGE is a logical next function to learn after SPAWN because it sends text to screen very easily, and MESSAGE has few methods beyond its main function:

MESSAGE:New(“This is a Message to everyone”, 15):ToAll()

Note that the message text could have been assigned to a variable, as in

MyMessage = “This is a Message to everyone”

MESSAGE:New( MyMessage, 15):ToAll()

## Core.SET

SET’s are similar to Lua tables and perform the important function of forming a single collection of like objects, for example GROUP’s, UNIT’s or ZONEs etc. A scripter can create a set of groups based on their:

* Name Prefix
* Coalition or Country
* Category (Airplanes, vehicles etc)
* Presence in a ZONE

Then s/he can manipulate the SET’s items, iterate (sequentially perform functions on) each item of the SET or count them. Consider the function

[SET\_GROUP.ForEachGroupCompletelyInZone](https://flightcontrol-master.github.io/MOOSE_DOCS_DEVELOP/Documentation/Core.Set.html#%23(SET_GROUP).ForEachGroupCompletelyInZone)

Determine how to check whether or not the set’s groups have arrived at a certain place. Sets are super powerful, and knowing group names is unnecessary, as the names can be obtained by iterating the set.

## Core.Zones

SPAWN section has many references to ZONES and this section examines in more detail the very powerful ZONE class in MOOSE. The ZONE class implements the use of scripting to control the behavior of DCS mission entities based on map location, for example spawning a group at a particular area. Zones can be created in the ME or directly in a script using the ZONE methods. A good first zone scripting exercise is creating a polygon zone, a process impossible with the ME. Refer to the polygon zone example in *Adapting An Existing Script.*

Example:

Zone\_Veh = GROUP:FindByName( "3R\_Zone\_Veh" ) -- Vehicle that defines the zone (3R\_Zone\_Veh)

Armor\_Zone = ZONE\_POLYGON:New( "Armor Zone", Zone\_Veh ) -- Instantiate Polyzone object

Zone\_Vec2 = Armor\_Zone:GetRandomVec2() -- Selects a random point in Armor\_Zone Polyzone

Red\_Armor = SPAWN:New( "3R\_Tank\_1" ) -- Instantiate a group based on 3R\_Tank\_1 characteristics

Red\_Armor:SpawnFromVec2( Zone\_Vec2 ) -- Spawns Red\_Armor at the random Vec2

## Core.SCHEDULER

Now, things get interesting, because the SCHEDULER class introduces the element of time to a script making it one of the most powerful classes in MOOSE. When called, a script runs once and is ignored thereafter; so, if a script action must occur repeatedly or after the initial script reading, like an aircraft entering a zone, the script must have a method to monitor mission progress over time to identify the conditions that trigger the action. Hence, on a schedule specified by the designer, the SCHEDULER periodically runs its set of actions (the SCHEDULER function). It can be configured to start after a selected mission time and then run on a specified frequency that is modified by a random factor, if desired.

The main function resembles a “for loop” in normal Lua scripting. A schedule can check virtually anything repeatedly. The following schedule runs after one second and again every ten seconds forever.

SCHEDULER:New( nil, function()

if … then

--do stuff

end

end,

{}, 1, 10 )

Check this method in the MOOSE docs:

[SCHEDULER:New(SchedulerObject, SchedulerFunction, SchedulerArguments, Start, Repeat,](https://flightcontrol-master.github.io/MOOSE_DOCS/Documentation/Scheduler.html#%23(SCHEDULER).New) [RandomizeFactor, Stop)](https://flightcontrol-master.github.io/MOOSE_DOCS/Documentation/Scheduler.html#%23(SCHEDULER).New)

The following schedule checks the mission every 5 seconds, after the first 15 seconds, to identify when the ME User Flag 10 == 1, then causes the unit named Fire Group to begin emitting red smoke. This is a simplified example that can be easily duplicated with ME triggers, but the available actions in MOOSE could be much more complex than anything reproducible with ME triggers.

Smoker = UNIT:FindByName( "Fire Group" )

GunScheduler = SCHEDULER:New( nil,

function()

if trigger.misc.getUserFlag( '10' ) == 1 then Smoker:SmokeRed()

end

end, {}, 15, 5

)

## Progress Recap

With the classes introduced thus far, the scripter has the tools to:

1. Spawn some tanks based on an ME tank group template.
2. Put them in a set based on their Prefix (names will start with Tanks#001 and increment)
3. Start a scheduler running every 10 seconds
4. Start a logic statement like if …then … else … end.
5. Count the number of units in a Zone
6. Iterate through the set and check heath
7. Print a message based on the logic
8. Take other actions (spawn more, spawn something different).
9. The Appendix and the MOOSE example missions have many examples of using SCHEDULER.

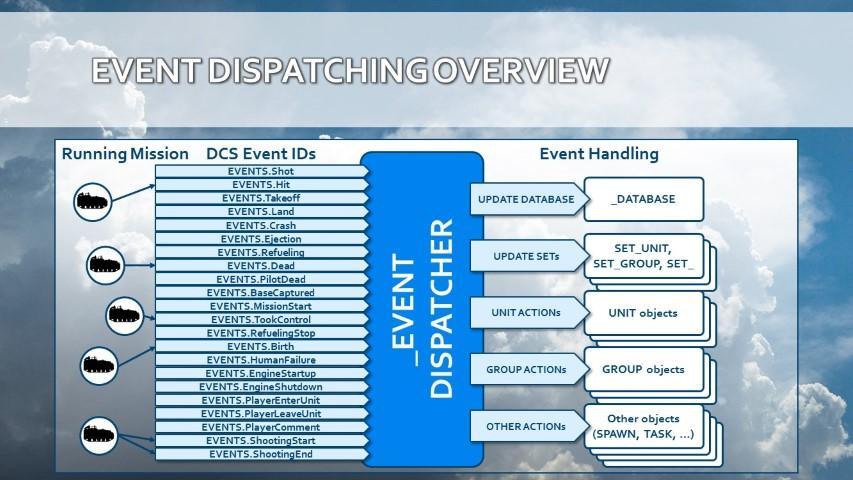
A couple of weeks should prove sufficient to become comfortable with these classes. The ability to create a series of mission actions using these methods indicates a good grasp of the basics.

# MOOSE Scripting 301

Now, with having familiarized with the core and MOOSE basics, examine the remaining wrappers, especially the GROUP Wrapper and the special methods unique to the UNIT Wrapper. Sets can consist of units or groups, and remembering the differences between the two classes will avoid many headaches. Other classes of significance in MOOSE Core include:

## Core.EVENT

The Event Handler class presents a second method to initiate mission actions after mission start, using the specific DCS recorded events, as described in the following graphic:



DCS writes these “events” to a table as they occur in real time as the mission progresses, including the event’s who, what and where. Shot, Hit, Takeoff, Land, Dead, Crash and Ejection are often used in scoring mechanisms; and the DCS mission debrief gets all its data from these event tables. The EVENT class methods, unlike SCHEDULER, monitor only the events that occur during the mission; and the EVENT function outputs a result when the event “fires”.

Be aware that in a MOOSE script a UNIT:EVENT gets handled for that unit only! Similarly, GROUP:EVENT gets handled only for all the units in that group. For all objects of other classes, like SET’s, the subscribed events get handled for all units within the mission!

BTRUnit:HandleEvent( EVENTS.Dead ) – Event fires only on BTRUnit death

BTRSet:HandleEvent( EVENTS.Dead ) – Event fires on EVERY unit death

## Core.Coordinate

One of the more complex classes, COORDINATE, introduces fine positioning control on the game map and empowers the scripter to calculate distances and bearings between points derived from zones and units, as well as obtaining weather and surface type, providing waypoints to groups, displacing (translating) coordinates, converting to various in-game readable formats like Bullseye, MGRS, LL DMS and putting smoke, explosions and markers on the ground or air. COORDINATE provides fine positioning like placing units, and it has randomization features, useful for random positioning of spawns using:

SpawnFromVec2()

### Vec2 and Vec3.

Vecs (2 and 3) define reference points on the mission map in 2 or 3 dimensional space in terms of x (E/W), y (up/down) and z (N/S) map directions.

### Coordinate.

COORDINATE is a MOOSE class that has methods not only to describe points in space, but also to manipulate the Vec point, get its attributes, get the characteristics of its physical environment and do things like emit smoke. A coordinate can be expressed in map terms like lat/long, MGRS, etc, in addition to x, y and z; and coordinates are directly obtainable from the cursor position shown on the bottom bar of the mission map.

### POINT\_VEC2 and POINT\_VEC3.

These COORDINATE class methods serve to change the values of a reference point, thus moving it to a different location.

## Wrapper.Controllable

Another complex class, CONTROLLABLE, enables direct control and tasking of groups.

## Ready Yet For Assistance?

A scripter at this stage is likely writing Lua, mixing in some core SSE API functions, doing more crazy eclectic stuff in an area of DCS with absolutely no feedback, and crashing the game to desktop! Ask for assistance on Discord, as this document cannot cover, in a reasonable number of paragraphs, all the nuances and pitfalls of the most complex MOOSE classes.

## Mission Environment Functional Classes

The last topic to cover in this User Guide is the MOOSE mission environment function classes that automate a great many complex mission activities. Examples include AIRBOSS, RAT, RANGE, MISSILE TRAINER, A2A\_Dispatcher, RADIO, DESIGNATE, DETECTION, ARTY, CARGO and WAREHOUSE. These class methods should not prove intimidating to a moderately experienced scripter, since they contain just a few lines of code; thus these classes do not appreciably advance one’s scripting competence, without detailed study of the script code, of course. Quite likely, a novice scripter who has reached this stage of the User Guide has already dabbled in these classes, as the temptation to tap their power would prove irresistible!

So far, this User Guide has been devoted to equipping a DCS mission designer to do basic scripting with MOOSE. The mission environment functional classes, however, typically consist of “off the shelf” entire libraries of code that the scripter can configure for a mission by customizing a few simple parameters. These classes can enhance a mission so profoundly that mission designers completely unfamiliar with Lua or MOOSE frequently attempt to employ them, leading to much confusion, frustration, help demands and anger management sessions. Using A2A\_Dispatcher, with RAT, CARGO, AIRBOSS, RANGE and MISSILE TRAINER/FOX, a scripter can make a great training map, but lacking basic scripting skills, s/he will have no idea how to make a table of group objects, save it to disk, spawn from tables, randomize events of large spawns, randomize their positions and so on.

## Final Summary

By following this User Guide to this point, a novice scripter should be able to read and understand most of the syntax in a MOOSE script, successfully used the SCHEDULER class, have written a few new scripts for his own missions and done a bit of troubleshooting. Script repair and log analysis skills probably still need considerable honing, however. Completing the following challenges permits one to claim the title of MOOSE Scripter 3rd Class:

* Written at least one simple function and reused it with at least one argument and a return value
* Constructed and referenced data from a table
* Used the SET:ForEachGroup() iterator
* Employed the SPAWN:OnSpawnGroup() function
* Used a “for loop”
* Referenced, without prompting, the MOOSE docs and the SSE API on Hoggit
* Found, understood and fixed script errors
* Used the DCS.log to identify a script error
* Obtained help to correct a mal-functioning script
* Created custom debugging tools and methods
* Abandoned a scripting objective beyond his ken
* Collected and saved script snippets
* Have a more complex mission scenario to script

These competencies indicate one can write MOOSE scripts in DCS, regardless of one’s scripting insecurities, so go boldly forth.

Reading other people’s work forms the path, now, for improving scripting skills. For example, read the code of Ciribob’s CTLD script and try to discern how it works. CTLD is somewhat long, very mature and complex; but it is structured and documented nicely for self-education purposes. Of course, CTLD was written with MiST environment, but that should present no difficulties at this stage, since many of the functions and methods will require researching the reference documents.

These following activities will serve to further enhance a mission designer’s MOOSE scripting skills:

* Read other people’s code and stuff linked on Discord.
* Try to troubleshoot other people’s issues, a skill more difficult than reading one’s own code.
* Study FunkyFranky’s Range script, a fairly simple script with some challenging little parts like tracking weapons.
* Download and examine missions from the ED User Files.
* Study popular short scripts, whatever you can get your hands on.
* Set some big goals for scripting projects and chip away at them.

# Advanced Troubleshooting

Script troubleshooting and debugging comprises **the single most important skill for improving Lua competency**, hands-down; and the DCS logs (refer to the next section) comprise the primary tool for understanding how a script malfunctioned and to learn script debugging. No one writes perfect code: all script writers make mistakes! Text copy-and-paste stands alone as the most common script-killer (followed closely by general stupidity, adult beverage imbibing, blindness, fatigue and dementia). Shadowze, a major contributor to the MOOSE community, has prepared an excellent guide to script troubleshooting, located here:

http://www.havoc-company.com/forum/viewtopic.php?f=30&t=1341

Before asking for help, please do read and understand all parts of this debugging guide, especially with respect to:

* Having a good log tailer with highlighting and watching your logs in real time
* Understanding the logs’ feedback, interpreting errors, how to deal with them
* Identifying common issues. “X is nil, function not found,” etc.
* Debugging a script with env.info, messages and similar
* Adept at serializing a table to discover its contents

Log tailor programs feature color-coding, search functions and other tools designed to make recorded log data easier to interpret. GamutLogViewer (low cost) and glogg (free: https://glogg.bonnefon.org/) are a couple of good ones.

## The DCS Log and Errors

The DCS.log, the primary script debugging tool, resides in: \Saved Games\DCS\Logs\dcs.log.DCS writes everything to this log as a mission loads and progresses. Observe that the log always has errors before the mission even runs, but these can be ignored! Log information of interest to MOOSE scripters begins way down the log with a line similar to this:

2020-04-16 23:44:34.375 INFO SCRIPTING: \*\*\* MOOSE STATIC INCLUDE START \*\*\*

Then follows the mission ‘registration’ of all the groups and units in the mission, followed by real time recording of mission events. Look first for the key word ERROR instead of INFO. An ‘error’ in programming parlance does not necessarily indicate a mistake. Rather, ‘error’ is a level of logging that displays text to a file, generally with the ‘log’ as a filetype. Logging levels vary according to each development studio and some generic examples include:

1. FATAL
2. ERROR
3. WARN
4. INFO
5. DEBUG
6. TRACE

DCS World will create env.info (an INFO log), env.warning (a WARNING log) and env.error (an ERROR log). Each of these log types outputs lines of text to the DCS.log. An env.error() halts the current program execution, whereas env.info does not. The log prints the event logging level, the source and then the output, i.e.

WARNING EDCORE <error detail>

INFO SOUND <error detail>

and finally, the one of interest to scripters during mission execution:

ERROR Scripting <we goofed here>

Errors with the error level of “Scripting” may halt the script execution, depending on what code block is running. In older versions of DCS, a message box popped up when this happened, and the entire game screeched to a halt! That behavior can still be enabled but no good reason exists to do so. Refer to:

<https://wiki.hoggitworld.com/view/DCS_func_setErrorMessageBoxEnabled>

A properly configured log tailer highlights errors via a regular expression or ‘regex’; and good practice dictates highlighting env.info’s by putting a keyword in all the error logging when writing the code. The log tailer can be configured to color-code these key words to identify the Script errors from other types.

Log text interpretation skills improve as more logs are examined and with accumulating experience in discerning important log events from ‘normal’ ones (didn’t we say that some errors are normal?). The more time spent with a log tailer open and scrolling in real time, the faster mission script errors will become apparent.

## Create a Test Mission

When a MOOSE script produces unexpected results, delete all the code, except for suspect sections and essential functions, and run the stripped-down script in a test mission consisting of only essential mission entities. Different sections of the script can be efficiently tested to isolate the misbehaving code; and the test script can be quickly modified and re-tested until it yields the expected results.

## Debugging

When a script misbehaves for an unknown reason, add extra lines of script code that output additional information into the logs. Serializing mitigates the limitation that only text can be output by making an unreadable datatype into a readable string. Refer to a serializing script example in the appendix. To discover the contents of a DCS table or something from MOOSE, look inside a SPAWN or ZONE object. Lua has built-in serializing functions, and the MiST imported serializing functions are available. Here is a custom serializing function for printing suspected faulty code to screen and logs:

functions(obj) -- obj == MOOSE object

local Result = routines.utils.oneLineSerialize(obj)

MESSAGE:New(Result,10):ToAll()

env.info(Result)

return Result

end

This function can take most objects and return them in a human readable form. It helps when you try to access something and get nil.

## How To Ask For Help

First, it is bad form to jump on MOOSE Discord, demand immediate assistance and criticize volunteered advice. The folks who hang out on Discord are not paid customer service professionals, but rather are hobbyists with families, obligations, their own projects and real lives. They offer their advice free of charge for the love of our hobby. Let’s examine some all-too-frequent questions on Discord (and the DCS forums):

*“My script doesn’t work. I want to do X and I explicitly followed the instructions, but it’s broken”*

The words, “broken” and “doesn’t work” are banned words, for they offer no meaningful information; and MOOSE troubleshooting skills do not include mind reading. The first analytical question is “what script?”, followed by “broken how?”

Another one:

*“Why doesn’t my script work?”*

[lists script snippet]

To such a question a reviewer will immediately respond with “What did DCS.log say*?”*, because the human eye reads code much less effectively than a computer (code is read and executed in a millisecond). In fact, reviewers concentrate first on the visual patterns and syntax, spelling and counting brackets, which strongly rely on how the text is formatted. In Discord, even using the markdowns, a reader still relies on how the author wrote the code, his whitespaces, bracketing habits and such. Use these markdowns in Discord:

```lua

[code]

```

They make code more readable. Even then, always upload the logs, preferably snipped around the error message. You did look, right? If you upload a mission, make sure it is free of mods. Even one of the pay maps will eliminate some potential assistance.

Another one:

*“I want to do X. Will someone write a script for me?”*

No, if a mission designer believes a script will enhance his mission, he must start learning to write script. This User Guide makes a good place to start, and all the MOOSE guardians do still try to encourage new scripters.

So the minimum requirement for a good request for help is: the script itself, preferably whole, unless one’s MOOSE competence allows providing an analyzable snippet, and the logs, preferably snipped to the only error in the logs that appears to be relevant, as well as the line number indicated in the provided script like so:

MyCode = Rubbish[0] 🡨 Line 147

The logs almost always accurately record a scripting error, stating the error clock time, script line number and a description of the error in ‘loggese’. Eliminating the error involves reading the logs with a log tailer, interpreting the ‘loggese’, identifying the line and taking corrective action.

Example errors:

2019-06-10 22:19:24.687 ERROR DCS: Mission script error: : [string "Spawn:New("test"):InitCleanUp(120):InitLimit(3,0):Spawn()"]:1: attempt to index global 'Spawn' (a nil value)

stack traceback:

[C]: ?

[string "Spawn:New("test"):InitCleanUp(120):InitLimit(3,0):Spawn()"]:1: in main chunk

The above example comes from a DO SCRIPT trigger, and the key error text reads:

()"]:1: attempt to index global 'Spawn' (a nil value)

* :1 -- Error lies in script Line 1.
* Attempt to index **global** -- meaning trying to use a global **function**
* ‘Spawn’ -- the item that Lua doesn’t understand
* (a nil value) -- A very common error. NIL is the absence of anything, i.e. nothing This signifies that the script has no idea what ‘Spawn’ is or means.

Now, the script Line 1 (the error line) reads:

*Spawn:New("test"):InitCleanUp(120):InitLimit(3,0):Spawn()*

So, either the first or last ‘Spawn’ could trigger the error. Reading the docs on SPAWN reveals that the case use is all UPPERCASE and ‘Spawn’ should have been ‘SPAWN’; thus, this is a typo caused by this author quickly typing into a do script (with no Intellisense warning of the mistake).

Another type of common error is this:

2019-06-10 21:08:18.536 ERROR Lua::Config: Call error onGameEvent:[string "./MissionEditor/modules/mul\_chat.lua"]:707: attempt to index field '?' (a nil value)

stack traceback:

[C]: ?

[string "./MissionEditor/modules/mul\_chat.lua"]:707: in function 'onGameEvent'

[string "./MissionEditor/GameGUI.lua"]:353: in function <[string "./MissionEditor/GameGUI.lua"]:350>

(tail call): ?

(tail call): ?.

The key words consist of

* :707: - - the first line on which the error was encountered, followed up by two other files with separate lines
* Attempt to index **field** ‘?’ (a nil value) – This is a classic table indexing issue. The script attempted to look at a table at the requested point and found nothing there, either due to an index error or no data existed at that location.
* :707: in function 'onGameEvent'
* :353: in function <[string "./MissionEditor/GameGUI.lua"]:350>

The last two lines show a “chain” of issues, in that the error propagated from one file, that pointed to another, that pointed to another. These could have been separate functions in the same script or in different files. As it happens, the error appears to result from an add-on hook in the ‘Hooks’ folder that is causing this error every time an EVENT fires. (we don’t have a solution as yet for this current error, but that is not important, since we have included it only for illustrative purposes)

**We sincerely hope you novice MOOSE scripters have found this User Guide useful and are encouraged to get out there and script!**

# Appendix

## History of MOOSE

Way back in the ancient “LockOn days”, Eagle Dynamics created some scripting API’s (programming interfaces) that permitted control of the mission’s behavior with some lines of Lua, a scripting language popular in computer games. These API’s existed solely for developer testing. They weren’t released as a customer tool and today they are unsupported with no official ED documentation. Boo!

Following the A-10C release, around the release date of DCS World in 2012, a couple of chaps called Speed and Grimes created a similar and now very mature higher level scripting framework called MiST, which is the father of MOOSE, and which greatly simplified DCS scripting. MiST and MOOSE have the same aims, but MOOSE is more comprehensive and expands functionality with complete modules. Some functions in MiST and MOOSE, for example spawning units, use completely different approaches. MiST still follows the requirement for a table for a spawn, whereas MOOSE uses a late-activated group to eliminate the table requirement. MOOSE has more features and has incorporated many of the MiST functions into its own framework. Fortunately, the two environments can happily cohabitate the same mission without noticeable impact.

Scroll forward to circa 2014. The author of MOOSE, Sven “FlightControl” found himself with a LOT of time on his hands and a very big vision. He sat down and wrote the early forms of MOOSE in an Object Oriented protocol. The Object Oriented development method uses a set of hierarchical objects in their own ordered classes that can interface with each other in a modular fashion. In early 2015 FlightControl released MOOSE to the DCS user community with scant fanfare. Development has continued, nevertheless, and by 2019, MOOSE had a large and busy Discord server with over 600 members, huge documentation and video libraries, several code contributors/champions, and currently sits at over 136 thousand lines of code and embedded documentation. More DCS mission developers are discovering the wonderful world of MOOSE every day.

## Snippets

Over time, MOOSE script writers began saving bits of scripts now called snippets, as a time-saving convenience. These script snippets perform a specific mission action and can be **judiciously** pasted into a developing script to avoid constantly reinventing the wheel. We recommend this practice for all MOOSE scripters. A few examples follow:

-----------------------------------------------------------------

-- ASSERT LOADFILE: dynamically loads a script file from a hard drive folder.

**assert**(**loadfile**("D:\\\_Google Drive\\DCS Missions\\file.lua"))()

---------------------------------------------------------------

--Copy route. Copies a route from one Group to another Group

**currentRoute** = **GROUP**:FindByName("groupTemplate"):CopyRoute(0, 0, **false**, 100)

**meGroup** = **GROUP**:FindByName("groupInME")

**meGroup**:Route(**currentRoute**)

---------------------------------------------------------------

--write data text to disk

**function** **writemissionfilestatus**(data)--defines the name of our save file and what we can write into it

**File** = **io**.open(**CFile**, "w")

**File**:write(data)

**File**:close()

**End**

-----------------------------------

--Check if a file exists

**function** **file\_exists**(name) --check if the file already exists for writing

**if** **lfs**.attributes(name) **then**

**return** **true**

**else**

**return** **false** **end**

**end**

---------------------------------------------------------------

--Check if file exists

**CFile** = "mySaveGame.txt"

**if** **file\_exists**(**CFile**) **then**

**dofile**(**CFile**)--execute the Lua inside the file

**env**.info("Campaign: Saved Campaign exists, loading from file...")

**trigger**.action.outText("Loading existing Campaign data from file...",4)

**else**

**writemissionfilestatus**("trigger.action.setUserFlag(1,1)") --writes Lua directly to file

**env**.info("Campaign: New campaign, writing new file...")

**dofile**(**CFile**) --also set the user flag whilst we are here.

**End**

---------------------------------------------------------------

-- Check if static is dead

**if** **StaticObject**.getLife(**StaticObject**.getByName('KOStatic1')) < 1 **then** **end**

---------------------------------------------------------------

-- Check airbase is blue by name

**function** **checkblue**(ABname)

**local** tmp = **coalition**.getAirbases(2)

**for** i=1,#tmp **do**

**if** tmp[i]:getName() == ABname **then**

**return** **true**

**end**

**end**

**end**

---------------------------------------------------------------

-- Count number of is alive in a set

**function** **AliveSet**(set) --thanks to CraigOwen for his help on this, he put me on the right path and it's down to him we stuck with it! #CommunityWin :)

**local** count = 0

set:ForEachClient(

**function** (setclient)

**if** setclient:IsAlive() **then**

count=count+1

**end**

**end**)

**return** count

**end**

---------------------------------------------------------------

-- Get the first alive group from a SPAWN and RTB it if not already

**GroupPlane1**, **Index1** = **ENEMY1**:GetFirstAliveGroup()

**if** **GroupPlane1** ~= **nil** **and** **GroupPlane1**:InAir() **then** **GroupPlane1**:RouteRTB() **end**

---------------------------------------------------------------

-- Spit out a few Random Spawns with a For loop

**local** ZoneA = **ZONE**:New( "Kobuleti")

**local** ZoneB = **ZONE**:New( "Batumi")

**local** ZoneC = **ZONE**:New( "Kutaisi")

**local** \_Num = 5

**local** \_Table={"Spawn1", "Spawn2", "Spawn3",}

**local** \_Zones={ZoneA, ZoneB, ZoneC,}

**Spawns** = {} --empty table for multiple spawns

**for** i=1,\_Num **do**

**local** tempRandGroup = **math**.random(1,\_Table.getn(\_Table))

**local** tempGrpAlias = "Spawn-" .. i

**Spawns**[i] = **SPAWN**:NewWithAlias(\_Table[tempRandGroup], tempGrpAlias):InitRandomizeZones(\_Zones)

**Spawns**[i]:Spawn()

**end**

---------------------------------------------------------------

-- Spawn from Static unit via Scheduler and go to random place in Zone when spawned

**local** \_Table={"Spawn1", "Spawn2", "Spawn3",}

**MyStatic** = **STATIC**:FindByName('MyStatic')

**MySpawn** = **SPAWN**:New('Spawn1')

:InitRandomizeTemplate( \_Table )

:InitLimit(6,0)--stops too many. make sure units in group no more than 6

**SCHEDULER**:New( **nil**,

**function**() --a new function for the scheduler to process

**do**

**MySpawn**:OnSpawnGroup(**function**( SpawnGroup)SpawnGroup:TaskRouteToZone( **ZONE**:New( "Kobuleti" ), **true**, 40, "Off Road" )**end**)

**MySpawn** = **STATIC**:FindByName('MyStatic')

**MySpawn**:SpawnFromStatic(**MyStatic**)--spawns the object 'test' from a static object called factory

**end**

**end**, {}, 5, 600, 0.5)

---------------------------------------------------------------

-- Scheduler and message, trigger flag and sound

**SCHEDULER**:New( **nil**,**function**()

**MESSAGE**:New("Checking for new units",10):ToAll()

**trigger**.action.outSound("micclick.ogg")

**Something** = **trigger**.misc.getUserFlag(1)

**end**, {}, 5, 600, 0.2 )

---------------------------------------------------------------

--scheduler and message, trigger flag and sound

**Player\_Unit = UNIT**:FindByName( "Player" ) -- Instantiates ME unit “Player”

**Expl\_Zone** = ZONE:New( "Boom Zone" ) -- Instantiates ME zone “Boom Zone”

**SCHEDULER**:New( nil,

**function()**

**if** Player\_Unit:IsInZone( Expl\_Zone ) then

**Explode** = COORDINATE:NewFromVec2( Expl\_Zone:GetRandomVec2() ) end

**end**, {}, 0, 4, .5

This little beauty sets off a ground explosion at a random location in an ME zone ( “Boom Zone” ) while the Player is in the zone. It starts immediately (time 0) when an ME trigger action starts the script and explosions occur on average every 4 seconds with a 50% time variation.

---------------------------------------------------------------

-- 3d Explosions getting closer each second (obsolete)

**function** **Bang**(plane)

**Outer** = 500

**Inner** = 250

**Zstart** = -500

**Ystart** = -100

**ExploStrength** = **math**.random(1,50)

**tgt** = **UNIT**:FindByName( plane )

**SCHEDULER**:New( **nil**, **function**()

**PlanePos** = **POINT\_VEC3**:NewFromVec3(**tgt**:GetVec3())

**RngPos** = **POINT\_VEC3**:NewFromVec3( **PlanePos**:GetRandomVec3InRadius(**Outer**, **Inner**) )

**AdjustedPos** = **RngPos**:AddX(0):AddY(**Ystart**):AddZ(**Zstart**)

**AdjustedPos**:Explosion(**ExploStrength**)

**if** **Outer** ~= 0 **then**

**Outer** = **Outer** - 10

**end**

**if** **Inner** ~= 0 **then**

**Inner** = **Inner** - 5

**end**

**if** **Zstart** ~= 0 **then**

**Zstart** = **Zstart** + 10

**end**

**if** **Ystart** ~= 0 **then**

**Ystart** = **Ystart** +5

**end**

**end**, {}, 0, 1, 1)

**end**

--------------------------------------------------------------

-- Barrage Flak (works but FLAK method of MiST is preferrable)

**BaseAltitude** = 4500 --in metres the lowest point of the barrage, or base altitude above ground

**FlakHeight** = 1500 --in metres, the total height of the Flak that it can fill above the base altitude. eg. 4500 and 1500 gives a volume of 4500-6000m

**Periodicity** = .7 --a value measured in seconds for which each flak burst will occur.

**Randomness** = .7 --a number between 0 and 1 that can change the Periodicity above. A value of 0 makes it observe the periodicity exactly. A value of 1 will randomise it up to double the time

**Strength** = 50 --the strength of the explosion. It's a bit difficult to explain but is about double the number of "lbs" of explosive

**MEflag** = 10 --the flag you are using to trigger flak to start and stop

**MEflagNo** = 1 -- the flag number of the flag you want to start the flak. When the flag is this number it starts, (until the flag is not that number, checked every Period

**Bursts** = 6 --the number of flak bursts per Period. Represents individual guns. Note, combined settings will have a performance effect

**flakzone** = **ZONE**:New("flak") --The zone where the flak will occupy. Zone name is "flak"

**function** **Flak**()

**local** alt = **math**.random(1,**FlakHeight**)

**local** rngpos = **flakzone**:GetRandomPointVec3()

**local** AdjustedPos = rngpos:AddX(0):AddY(**BaseAltitude** + alt):AddZ(0)

AdjustedPos:Explosion(**Strength**)

**end**

**SCHEDULER**:New( **nil**, **function**()

**local** Flag = **trigger**.misc.getUserFlag(**MEflag**) --looks for FLAG number being set for the trigger. Use the ME to make a coaltition in zone or something else.

**if** Flag == **MEflagNo** **then**

--MESSAGE:New("Flak should be running",10):ToAll()

**i** = 0

**while** **i** < **Bursts**

**do**

**Flak**()

**i**=**i**+1

**end**

**else**

--nothing

**end**

**end**, {}, 0, **Periodicity**, **Randomness**)

---------------------------------------------------------

-- Serialize table to message and logs

**function** **s**(table)

**local** Result = **routines**.utils.oneLineSerialize(table)

**MESSAGE**:New(Result,10):ToAll()

**env**.info(Result)

**return** Result

**end**

---------------------------------------------------------

-- Check units by name in SET

**SU25TSETClient**:ForEachClient(**function** (MooseClient)

**exists** = **CheckClientExistsInSet**(MooseClient:GetName(),**clientName**)

**end**)

**CheckClientExistsInSet** = **function** (client,searchString)

**if** client == searchString **then**

**return** **true**

**else**

**return** **false**

**end**

**end**

--------------------------------------------------------------

-- Get player and group and units from event

**DeleteLanding** = **EVENTHANDLER**:New()

**DeleteLanding**:HandleEvent( **EVENTS**.Land )

**function** **DeleteLanding**:OnEventLand( EventData )

**ThisGroup** = **GROUP**:FindByName(EventData.IniGroupName)

**GroupUnit** = **ThisGroup**:GetDCSUnit(1)

**FirstUnit** = **UNIT**:Find(**GroupUnit**)

**if** **FirstUnit**:GetPlayerName() **then**

**PlayerName** = **FirstUnit**:GetPlayerName()

**env**.info(**PlayerName** .. " has landed")

**else**

**env**.info("Not a player landed, deleting")

**ScheduleDelete**(**ThisGroup**)-- custom schedule to delete a group

**end**

**end**

--------------------------------------------------------------

-- Match a string

**if** **EventData**.IniGroupName:match("Rescue Helo") **then**

**env**.info("Leave my Helo alone.")

**else**

**end**

------------------------------

-- Spawn when airbase is a coalition

**function** **getAB**(airbaseName)

**local** TmpAB = **AIRBASE**:FindByName(airbaseName)

**if** TmpAB:GetCoalition() == 1 **then**

**return** "Red"

**elseif** TmpAB:GetCoalition() == 2 **then**

**return** "Blue"

**elseif** TmpAB:GetCoalition() == 0 **then**

**return** "Neutral"

**else** **return** "Not an Airbase"

**end**

**end**

**MySpawn** = **SPAWN**:New("bKobCAS1"):InitLimit(1,0):SpawnScheduled(500,.2)

**MySpawn**:SpawnScheduleStop()

**SCHEDULER**:New( **nil**,

**function**()

**if** **getAB**("Batumi") == "Red" **then**

**MySpawn**:SpawnScheduleStart()

**else**

**MySpawn**:SpawnScheduleStop()

**end**

**end**, {}, 15, 30)

--------------

-- TABLE SAVE

-- http://lua-users.org/wiki/SaveTableToFile

**local** **function** exportstring( s )

**return** **string**.format("%q", s)

**end**

--// The Save Function

**function** **table**.save( tbl,filename )

**local** charS,charE = " ","\n"

**local** file,err = **io**.open( filename, "w+" ) --edited

**if** err **then** **return** err **end**

-- initiate variables for save procedure

**local** tables,lookup = { tbl },{ [tbl] = 1 }

file:write( "return {"..charE )

**for** idx,t **in** **ipairs**( tables ) **do**

file:write( "-- Table: {"..idx.."}"..charE )

file:write( "{"..charE )

**local** thandled = {}

**for** i,v **in** **ipairs**( t ) **do**

thandled[i] = **true**

**local** stype = **type**( v )

-- only handle value

**if** stype == "table" **then**

**if** **not** lookup[v] **then**

**table**.insert( tables, v )

lookup[v] = #tables

**end**

file:write( charS.."{"..lookup[v].."},"..charE )

**elseif** stype == "string" **then**

file:write( charS..exportstring( v )..","..charE )

**elseif** stype == "number" **then**

file:write( charS..**tostring**( v )..","..charE )

**end**

**end**

**for** i,v **in** **pairs**( t ) **do**

-- escape handled values

**if** (**not** thandled[i]) **then**

**local** str = ""

**local** stype = **type**( i )

-- handle index

**if** stype == "table" **then**

**if** **not** lookup[i] **then**

**table**.insert( tables,i )

lookup[i] = #tables

**end**

str = charS.."[{"..lookup[i].."}]="

**elseif** stype == "string" **then**

str = charS.."["..exportstring( i ).."]="

**elseif** stype == "number" **then**

str = charS.."["..**tostring**( i ).."]="

**end**

**if** str ~= "" **then**

stype = **type**( v )

-- handle value

**if** stype == "table" **then**

**if** **not** lookup[v] **then**

**table**.insert( tables,v )

lookup[v] = #tables

**end**

file:write( str.."{"..lookup[v].."},"..charE )

**elseif** stype == "string" **then**

file:write( str..exportstring( v )..","..charE )

**elseif** stype == "number" **then**

file:write( str..**tostring**( v )..","..charE )

**end**

**end**

**end**

**end**

file:write( "},"..charE )

**end**

file:write( "}" )

file:close()

**end**

--// The Load Function

**function** **table**.load( sfile )

**local** ftables,err = **loadfile**( sfile )

**if** err **then** **return** **\_**,err **end**

**local** tables = ftables()

**for** idx = 1,#tables **do**

**local** tolinki = {}

**for** i,v **in** **pairs**( tables[idx] ) **do**

**if** **type**( v ) == "table" **then**

tables[idx][i] = tables[v[1]]

**end**

**if** **type**( i ) == "table" **and** tables[i[1]] **then**

**table**.insert( tolinki,{ i,tables[i[1]] } )

**end**

**end**

-- link indices

**for** \_,v **in** **ipairs**( tolinki ) **do**

tables[idx][v[2]],tables[idx][v[1]] = tables[idx][v[1]],**nil**

**end**

**end**

**return** tables[1]

**end**

-- close do

**function** **table**.val\_to\_str ( v )

**if** "string" == **type**( v ) **then**

v = **string**.gsub( v, "\n", "\\n" )

**if** **string**.match( **string**.gsub(v,"[^'\"]",""), '^"+$' ) **then**

**return** "'" .. v .. "'"

**end**

**return** '"' .. **string**.gsub(v,'"', '\\"' ) .. '"'

**else**

**return** "table" == **type**( v ) **and** **table**.tostring( v ) **or**

**tostring**( v )

**end**

**end**

**function** **table**.key\_to\_str ( k )

**if** "string" == **type**( k ) **and** **string**.match( k, "^[\_%a][\_%a%d]\*$" ) **then**

**return** k

**else**

**return** "[" .. **table**.val\_to\_str( k ) .. "]"

**end**

**end**

**function** **table**.tostring( tbl )

**local** result, done = {}, {}

**for** k, v **in** **ipairs**( tbl ) **do**

**table**.insert( result, **table**.val\_to\_str( v ) )

done[ k ] = **true**

**end**

**for** k, v **in** **pairs**( tbl ) **do**

**if** **not** done[ k ] **then**

**table**.insert( result,

**table**.key\_to\_str( k ) .. "=" .. **table**.val\_to\_str( v ) )

**end**

**end**

**return** "{" .. **table**.concat( result, "," ) .. "}"

**end**

----------------------------------

-- Integrated Serialise With Cycles

**function** **IntegratedbasicSerialize**(s)

**if** s == **nil** **then**

**return** "\"\""

**else**

**if** ((**type**(s) == 'number') **or** (**type**(s) == 'boolean') **or** (**type**(s) == 'function') **or** (**type**(s) == 'table') **or** (**type**(s) == 'userdata') ) **then**

**return** **tostring**(s)

**elseif** **type**(s) == 'string' **then**

**return** **string**.format('%q', s)

**end**

**end**

**end**

-- imported slmod.serializeWithCycles (Speed)

**function** **IntegratedserializeWithCycles**(name, value, saved)

**local** basicSerialize = **function** (o)

**if** **type**(o) == "number" **then**

**return** **tostring**(o)

**elseif** **type**(o) == "boolean" **then**

**return** **tostring**(o)

**else** -- assume it is a string

**return** **IntegratedbasicSerialize**(o)

**end**

**end**

**local** t\_str = {}

saved = saved **or** {} -- initial value

**if** ((**type**(value) == 'string') **or** (**type**(value) == 'number') **or** (**type**(value) == 'table') **or** (**type**(value) == 'boolean')) **then**

**table**.insert(t\_str, name .. " = ")

**if** **type**(value) == "number" **or** **type**(value) == "string" **or** **type**(value) == "boolean" **then**

**table**.insert(t\_str, basicSerialize(value) .. "\n")

**else**

**if** saved[value] **then** -- value already saved?

**table**.insert(t\_str, saved[value] .. "\n")

**else**

saved[value] = name -- save name for next time

**table**.insert(t\_str, "{}\n")

**for** k,v **in** **pairs**(value) **do** -- save its fields

**local** fieldname = **string**.format("%s[%s]", name, basicSerialize(k))

**table**.insert(t\_str, **IntegratedserializeWithCycles**(fieldname, v, saved))

**end**

**end**

**end**

**return** **table**.concat(t\_str)

**else**

**return** ""

**end**

**end**

--------------------------------------------

--[[ FUNCTION EDITED FROM GRIMES https://forums.eagle.ru/showthread.php?t=242343

Gets the pair of airbases closest to each other of opposite coalitions

Requires Mist but could be converted. --]]

**function** **getNearest**()

**local** ab = **world**.getAirbases()

**nearestPair** = {dist = 1000000} -- assign large value to start so that almost anything will be smaller than it

**for** i = 1, #ab **do**

**local** coa = (ab[i]):getCoalition()

**local** pos = (ab[i]):getPoint()

**if** coa ~= 0 **then** -- not neutral

**for** j = 1, #ab **do**

**local** oCoa = **Airbase**.getCoalition(ab[j])

--TO DO MUST also exclude Ships.

**if** i ~= j **and** (coa ~= oCoa **and** oCoa ~= 0 ) **then** -- not the same airbase and not the same coalition and not neutral

**local** dist = **mist**.utils.get2DDist(pos, (ab[j]):getPoint()) -- whatever you wanted to use to get the distance between the two points

**if** dist < **nearestPair**.dist **then** -- check against the shortest distance between opposing side airbases

**nearestPair** = {dist = dist, bases = {i, j}} -- newly defined entry

**end**

**end**

**end**

**end**

**end**

**if** **not** **nearestPair**.bases **then**

**env**.error("No airbases found as a pair!")

**return** **false**

**else**

**return** **nearestPair**.bases

**end**

**end**

-----------------------------

-- Execute a restart batch file based on real server time, from inside mission

**SCHEDULER**:New( **nil**, **function**()

**REBOOT**=**false**

**Reboot1** = 2350

**Reboot2** = 0700

**ServerTime** = **os**.date("%H:%M")

**NumberTime** = **os**.date("%H%M")

**env**.info(**os**.date("The server's local time is " .. **ServerTime**))

**MESSAGE**:New("The Server's local time is " .. **ServerTime**):ToAll()

**MESSAGE**:New("The Server will reboot at 23:30 in " .. **tonumber**(**NumberTime**)-0010 .. "mins."):ToAll()

**if** **tonumber**(**NumberTime**) == 0010 **then**

**MESSAGE**:New("The Server is rebooting"):ToAll()

--os.execute[[C:\\Users\thebg\\Desktop\\restart.bat]]

**REBOOT**=**true**

**end**

**end**, {},2, 10)

------------------------------

-- Event Handler for scenery

**EventHandler** = **EVENTHANDLER**:New()

**EventHandler**:HandleEvent( **EVENTS**.Dead )

**function** **EventHandler**:OnEventDead( EventData )

-- POTI EAST

**if** EventData.IniUnitName == 74645955 **then** --Scenery has a IniUnitName. You can find it by blowing it up first and catching the ID

**MESSAGE**:New("Poti East Bridge destroyed",10):ToAll()

**flag1** = 1

**trigger**.action.outSound("radio1.ogg")

**elseif** EventData.IniUnitName == 74645956 **then**

**MESSAGE**:New("Poti East Bridge destroyed",10):ToAll()

**flag1** = 1

**trigger**.action.outSound("radio1.ogg")

**end**

-- You can do loads more

**end**

**SCHEDULER**:New( **nil**, **function**()

**if** **flag1**==1 **then**

**BridgesDown** = 1

**MESSAGE**:New("All the bridges near Poti have now been destroyed, we can focus on the airfield now.",10):ToAll()

**trigger**.action.outSound("radio1.ogg")

**end**

**end**, {}, 60, 30, 0)

---------------------------

-- Very Basic IADS (simplistic)

**SamSet1** = **SET\_GROUP**:New():FilterPrefixes("ABsam KOB"):FilterStart()

**SamSet1**:ForEachGroup(

**function**( MooseGroup1 )

**local** chance = **math**.random(1,99)

**if** chance > 10 **then**

MooseGroup1:OptionAlarmStateRed()

**else**

MooseGroup1:OptionAlarmStateGreen()

**end**

**end**, {}, 1, 30)

---------------------------------

-- Getting tanks to randomly move between zones continuously

**local** ZoneList1 = {

**ZONE**:New( "ZONE1" ),

**ZONE**:New( "ZONE2" ),

**ZONE**:New( "ZONE3" ),

**ZONE**:New( "ZONE4" ),

**ZONE**:New( "ZONE5" ),

}

**function** **ReRouteR**( VehicleGroup )

**local** ZoneNumber = **math**.random( 1, #ZoneList1 )

**local** FromCoord = VehicleGroup:GetCoordinate() -- Core.Point#COORDINATE

**local** FromWP = FromCoord:WaypointGround()

**local** ZoneTo = ZoneList1[ ZoneNumber ] -- Core.Zone#ZONE

**local** ToCoord = ZoneTo:GetCoordinate()

**local** ToWP = ToCoord:WaypointGround( 50, "On Road" )

**local** TaskReRoute = VehicleGroup:TaskFunction( "ReRouteR" )

VehicleGroup:SetTaskWaypoint( ToWP, TaskReRoute )

VehicleGroup:Route( { FromWP, ToWP }, 4 )

**end**

**Rtanks** = **SPAWN**

:New("rtanks")

:InitLimit(3,0)

**Rtanks**:OnSpawnGroup(**function**( SpawnGroupR )**ReRouteR**(SpawnGroupR)**end**)

:SpawnScheduled(100,0)