

# MOOSE User Guide



By MOOSE contributor team v1.0

## Table of Contents

<b>Document Purpose and Aim.....</b>	<b>1</b>
<b>The different paths.....</b>	<b>2</b>
<b>How to Identify which parts to read. Who am i?.....</b>	<b>3</b>
Beginner.....	3
Intermediate.....	3
Advanced.....	4
<b>Beginners start here.....</b>	<b>5</b>
What is a Script?.....	5
What is MOOSE?.....	5
What Can MOOSE Do For Me?.....	5
What If I Don't Want To Learn Scripting?.....	5
How Do I Get and Install MOOSE?.....	5
How Do I Install MOOSE Scripts?.....	6
How do I write a Script?.....	7
What are my DCS Logs?.....	7
Where are my DCS Logs?.....	7
How can I read my DCS Log?.....	7
Where can I find example MOOSE missions?.....	7
Where is the documentation for MOOSE classes?.....	8
How to ask for help.....	8
<b>Intermediate scripting.....</b>	<b>9</b>
<b>Bookmarks &amp; References (Intermediate).....</b>	<b>10</b>
Where Do I Find MOOSE Scripts?.....	11
MOOSE Documentation.....	11
Checkpoint reached:.....	11
<b>Configuring an IDE like LDT / VSC (Intermediate).....</b>	<b>12</b>
Checkpoint reached:.....	12
<b>Logs, Troubleshooting &amp; Debugging (Intermediate).....</b>	<b>13</b>
Log tailing.....	13
The DCS Log and Errors.....	14
Debugging.....	15
How To Ask For Help.....	16
Checkpoint reached:.....	18
<b>Lua 101 (Intermediate).....</b>	<b>19</b>
Essential Scripting Principles.....	19
Lua Essentials - Datatypes.....	19

Now read Ch1-4 .....	23
Checkpoint reached:.....	23
<b>The MOOSE Framework Overview .....</b>	<b>24</b>
The DCS Simulator Scripting Engine (SSE) .....	25
Running Scripts in DCS.....	25
Running scripts directly from Hard Drive .....	26
Checkpoint reached:.....	26
<b>MOOSE 101 - SPAWN (Intermediate) .....</b>	<b>27</b>
Anatomy of a MOOSE Method .....	28
Next steps .....	31
Checkpoint reached.....	32
<b>MOOSE Documentation and Object Oriented structure (Intermediate) .....</b>	<b>33</b>
Inheritance in Object Oriented Programming .....	33
What is a Core Class.....	34
What is an Object?.....	34
What is a Wrapper?.....	34
Functional, Tasking, AI, Ops, Utils etc.....	35
How to read the Documentation .....	35
Checkpoint reached.....	38
<b>MOOSE Scripting 201 (Intermediate).....</b>	<b>39</b>
Wrapper.GROUP .....	39
Core.MESSAGE .....	39
Core.SET .....	39
Core.Zones .....	40
Core.SCHEDULER .....	40
Progress Recap .....	41
<b>MOOSE Scripting 301 (Intermediate).....</b>	<b>43</b>
Core.EVENT .....	43
Core.Coordinate.....	44
Wrapper.Controllable.....	44
Mission Environment Functional Classes.....	44
Final Summary .....	45
Checkpoint reached.....	46
<b>Advanced MOOSE .....</b>	<b>47</b>
<b>Appendix A - How DCS Works .....</b>	<b>48</b>
Evolution of DCS SSE.....	48
Script Execution Scope.....	48
Anatomy of The Mission (.miz) File.....	49
The Mission.miz\Mission Table .....	50

<i>Limitations .....</i>	<i>52</i>
<i>Appendix B - History of MOOSE .....</i>	<i>54</i>
<i>Appendix C - Troubleshooting techniques.....</i>	<i>55</i>
<i>Appendix D - Snippets.....</i>	<i>57</i>

### Document Purpose and Aim

This companion guide answers the question, "*How do I begin to use MOOSE*". All required links and tools are provided from beginner up to Developer skill. The purpose is to reduce user frustration, as well as to facilitate support and self-help.

The Guide covers all the individual elements needed to produce a MOOSE script and is quite broad. It doesn't duplicate available scripting documentation but does provide you the tools and locations of such documentation, along with helpful pointers.

This document has been written and ratified by the team that maintains MOOSE. We hope it answers questions you may have regarding MOOSE scripting, as well as encourage you to get out there and write great scripts!

### The different paths

This guide aims to serve *three different types* of use cases. These are called “entry points”.

The “**Beginner**” entry point to learning MOOSE, is for those who have no interest in learning a programming language and see MOOSE as a way to cut down the time they need to spend on mission design, not increase it. This would include making best use of the modules like RAT, Range, Dispatchers, Airboss, Fox. This is not necessarily people new to everything around Lua and scripting, but people investing the least time for the most gain.

The “**Intermediate**” starting point is for people that have a positive desire to learn more scripting and are willing to invest some time for a medium term goal. They will be looking at writing small scripts using several full classes going together, with some Core MOOSE functions. These people will explore more and are open to small scripts and building upon that.

The “**Advanced**” starting point is for confident people that can already understand most of the key elements of scripting and are mostly looking at developing new scripts using Core MOOSE, rather than using the large classes. They may not want to help out in the community, but they probably have some experiences to offer other people.

Does learning MOOSE mean you will learn Lua? Do you need to learn Lua to use MOOSE? Well, MOOSE and Lua are so entwined that in the process of leaning MOOSE you will also absorb a good deal of Lua naturally. But you don't need to learn Lua to begin using MOOSE. In fact, MOOSE helps you learn basic Lua faster, and without you realising it is happening.

# How to Identify which parts to read. Who am i?

## Beginner

*"I just want to copy and paste stuff. I do not want to learn how to script. I want to use modules like Dispatcher, Airboss and RAT, because I like what they do. I generally consume these with little understanding, and I won't be attempting to learn anything beyond how to get it going, basic tweaking and asking for help.*

**This path concentrates on finding the documentation, where to download Moose.lua, how to find your logs and ask for help properly, as well as minimum tools required. Following this path avoids wasted time setting up Lua Development tools (LDT) and getting lost in videos, but adds "how to ask for help", because self-help is a primary learning objective.**

**Tools:** Notepad++ (NPP), Baretail log tailer, Moose.lua, bookmarking links to the MOOSE demo missions and MOOSE documentation, in your browser.

**Starting Skills: Basic DCS mission Editor skills, reasonable PC/Windows Skills.**

**Process:** Install NPP, and install Baretail. Then download Moose.lua and go to the MOOSE demo missions and download and launch and tweak. Read the module documentation for extra configuration.

**Suggested content:** Airboss, RAT, A2A\_Dispatcher, RANGE, FOX and full modules that require little or well documented configuration.

**Limitations/when to move on:** You won't have Intellisense (a script environment feature that provides MOOSE-specific auto-complete); and you probably cannot progress further than simple lines of SPAWN code until you have. You will need to install LDT to learn more advanced MOOSE scripting, using Intellisense; and you will have to learn how to debug, serialise and log-tail next.

## Intermediate

*I want to write short scripts mainly to spawn things, make messages on screen, use timers and make groups do things. I am open to learning more but want to start simple. I will need help finding out how to write basic code, but I do not want to be intimidated from the outset.*

**Tools:** Lua Development Tools as an IDE, Log Tailer, Moose.lua static version, bookmark links to the MOOSE demo missions and MOOSE documentation.

**Starting Skills: Familiar with running scripts in Mission Editor, can create a basic script to start with and find the documentation and check logs for errors.**

**Process:** Install & configure LDT (or optionally Visual Studio Code, VSC), install Baretail, download Moose.lua. then go to the MOOSE demo missions, download them and try them out for ideas.

**Suggested content:** Look at SPAWN as a class, use and understand Wrappers, complete this guide's contents and full module demos (AIRBOSS, RAT, Dispatchers etc) and download and tweak them as desired. Read the documentation so you get an idea what they do. Will now begin experimenting and seeing what happens in the logs and this will be how to learn from now on.

**Limitations/when to move on:** This segment takes a long time. Expanding your knowledge is an ongoing process. The signs that you have grown beyond intermediate is that you find most of your answers on the StackExchange web site, rather than the documentation. You are learning more about Lua itself than MOOSE, you occasionally slip into SSE Lua to test out MOOSE and you find the error in MOOSE pretty quickly.

### Advanced

*"I am familiar with scripting, I need to use a proper integrated development environment (IDE) and tackle learning MOOSE with familiar tools. I am an adept self trouble-shooter and only need a few pointers. I am going to continue writing my own custom scripts using the core of moose "*

**Tools:** LDT configured with Intellisense and debugger, Github (GH) Desktop, signed up to GH as a contributor, log tailer, Moose.lua static or dynamic version, MOOSE demo missions, MOOSE documentation, Hoggit Wiki, Lua documentation (esp ch1-4), Troubleshooting guide

**Starting Skills:** Basic scripting/programming in another language, familiar with programming concepts, understands limits of the SSE, understand the usage of an IDE, familiar with GH, troubleshoot their own code, beginning to troubleshoot other peoples' code.

**Process:** Setup LDT or VSC and a debugger as well as a log tailer. Setup your workflow to run mission scripts directly from disk Read through the Hoggit Wiki to understand the limitations of the DCS SSE. Read first four chapters of Lua online manual to understand Lua's main differences, Learning MOOSE from the CORE classes - WRAPPERS(GROUP, UNIT, STATIC), MESSAGE, SCHEDULER, SET, ZONE, EVENT, POINT, CONTROLLABLE

**Suggested content:** Using MOOSE CORE classes over modules. Using core Lua like table manipulation.



### Beginners start here

This very short chapter is for people identifying as a consumer of Moose and not wishing to learn to script. This is a condensed FAQ and set of links to get you up and running. It specifically avoids any complexity.

### What is a Script?

A *script* is a plain text set of instructions read by a computer and processed. Scripts are often used to interface between a complex compiled language and a simpler language. For example, we write batch files to Windows because DOS was written in a compiled language. DOS helps us navigate a computer file system, via a script.

DCS has a Simulator Scripting Engine (SSE) which acts as an interface between the DCS application and the Lua scripting language.

### What is MOOSE?

**M**ission **O**bject **O**riented **S**cripting **E**nvironment, is a scripting framework that attempts to make the scripting of Lua instructions to the SSE, easier, simpler and shorter. You write a script, using MOOSE's words, in Lua to the SSE, which talks to DCS, which makes things happen in the game. It is over 5MB of code, with as many words as the Bible and the core of it was written over several years by one person. MOOSE is the brain-child of an extremely talented programmer; 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?

Whilst MOOSE can be used to write customised Lua scripts, you are probably not caring for learning Lua right now. Instead, what Moose can do for you is allow you to use a script written by someone in MOOSE and you can just configure its basic settings to work in your mission. Such things as Airboss, handling carriers and ATC, A2A\_Dispatcher, that creates a network of fighter CAPs and GCI, RAT, which makes easy random air traffic, Range, which counts hits on targets so you can practice, FOX missile trainer so you don't get killed by SAM's, and so on. You will need to look through examples to know what you want, but if you found this document, you know you need something you saw, right?

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

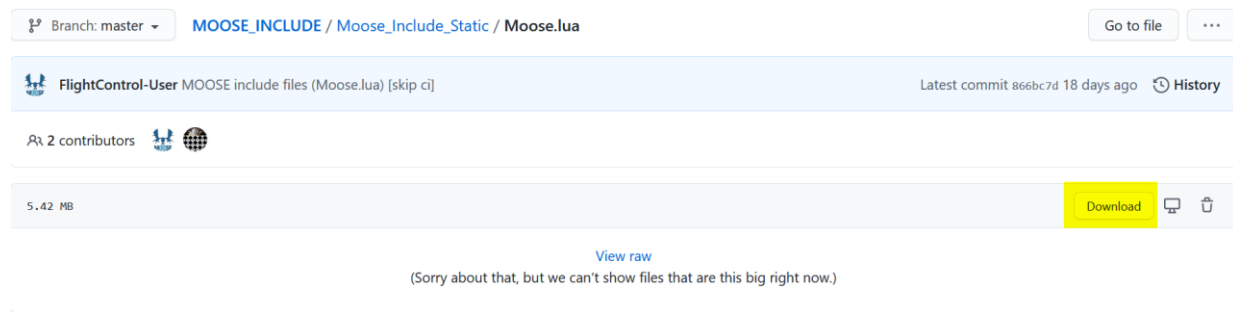
You don't need to. This is your chapter. All you need to do is go to the demo missions repository, pull out a demo mission, try it out, edit the parameters to suit in Notepad and if you want more scripts, combine them all up! Look at the demos here:

[https://github.com/FlightControl-Master/MOOSE\\_MISSIONS](https://github.com/FlightControl-Master/MOOSE_MISSIONS)

### How Do I Get and Install MOOSE?

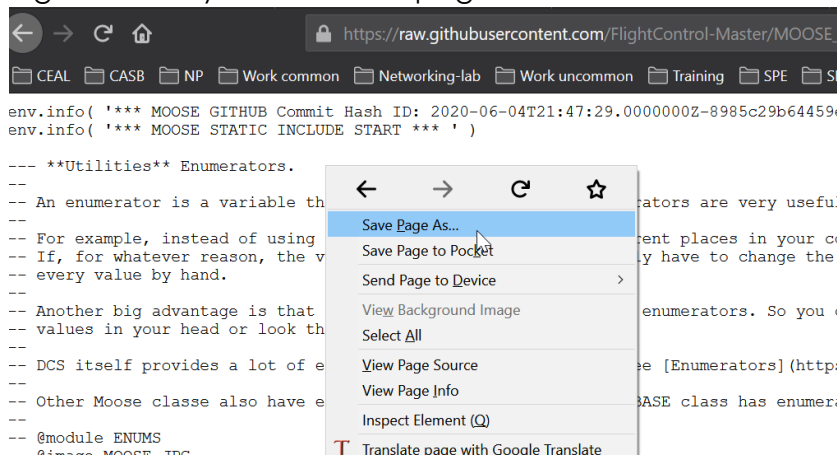
MOOSE is just a single file called **Moose.lua**. Multiple versions exist, but you only need to worry about the Moose.lua from the stable branch.

[https://github.com/FlightControl-Master/MOOSE\\_INCLUDE/blob/master/Moose\\_Include\\_Static/Moose.lua](https://github.com/FlightControl-Master/MOOSE_INCLUDE/blob/master/Moose_Include_Static/Moose.lua)



**WARNING!** GitHub is not user friendly when downloading large files, so, to download Moose.lua:

- Click on Moose.lua at the [link](#). Invokes a new window
- Click on Download button on the right. You will see the contents of Moose.lua but not get the file!
- Right-click anywhere on the page. This invokes a selection list.



- Click on **“Save Page As”**. This invokes a Save dialogue window.
- Select desired folder, Click Save.

You can download Moose.lua from the develop branch also, and if you search the “Include” folder you can find stripped down versions without documentation (Moose\_.lua). You can identify the version of MOOSE by the date it was created.

## How Do I Install MOOSE Scripts?

The 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.

## How do I write a Script?

You notice we need the Moose.lua script loaded before ours. You can write your script in any text editor to begin with, but as a brand new script writer, please use Notepad++ (NPP) which can be found here <https://notepad-plus-plus.org/downloads/>

## What are my DCS logs?

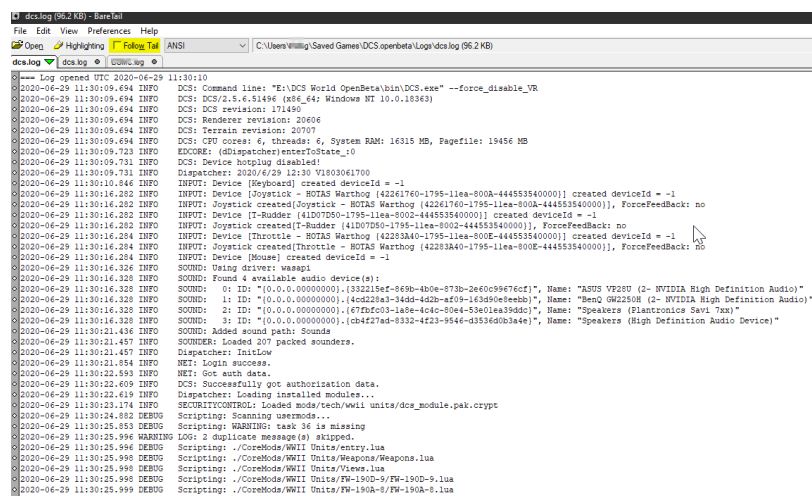
The DCS log is a super important and useful log for the entire of DCS World. All scripting and other errors are recorded here. It is the one stop shop for things that occurred in your mission. They will tell you first if there was a mistake.

## Where are my DCS logs?

Your Saved Games folder has a folder with the name of your install. It can be "DCS" or "DCS.openbeta" or something else, so beware if you have messed about with folder names in the past. C:\Users\<USERNAME>\Saved Games\DCS.openbeta\Logs\**dcsl.log**

## How can I read my DCS log?

<https://www.baremetalsoft.com/baretail/> Baretail is a tiny free program will update in real time as your mission progresses; so not only can you follow your logs as they happen, but they highlight errors which can be very helpful in understanding where you messed up. Glogg also does the same thing and you can get it here <https://glogg.bonnefon.org/> You can also tail in Notepad++. We only recommend a tailer because actually you get to learn a lot about DCS just having a glance at the logs. And this helps everything!



```

Log opened UTC 2020-06-29 11:30:10
2020-06-29 11:30:09.694 INFO DCS: Command line: "E:\DCS World OpenBeta\bin\DCS.exe" --force_disable_VR
2020-06-29 11:30:09.694 INFO DCS: DCS 2.5.6.31496 (ref_44; Windows NT 10.0.18363)
2020-06-29 11:30:09.694 INFO DCS: DCS revision: 171490
2020-06-29 11:30:09.694 INFO DCS: Renderer revision: 20406
2020-06-29 11:30:09.694 INFO DCS: Terrain revision: 20707
2020-06-29 11:30:09.694 INFO DCS: CPU cores: 6, threads: 6, System RAM: 16315 MB, Pagefile: 19456 MB
2020-06-29 11:30:09.723 INFO EDCORE: (dispatcher)enterToState:0
2020-06-29 11:30:09.731 INFO DCS: Device hotplug disabled!
2020-06-29 11:30:09.731 INFO Dispatcher: 2020/6/29 12:30 V1803061700
2020-06-29 11:30:10.046 INFO INPUT: Device [Keyboard] created deviceId = -1
2020-06-29 11:30:16.202 INFO INPUT: Device [Joystick - HOTAS Warthog (42261760-1795-11ea-800a-444553540000)] created deviceId = -1
2020-06-29 11:30:16.202 INFO INPUT: Joystick created[Joystick - HOTAS Warthog (42261760-1795-11ea-800a-444553540000)], ForceFeedback: no
2020-06-29 11:30:16.202 INFO INPUT: Device [T-Rudder (41007050-1795-11ea-8002-444553540000)] created deviceId = -1
2020-06-29 11:30:16.202 INFO INPUT: Joystick created[T-Rudder (41007050-1795-11ea-8002-444553540000)], ForceFeedback: no
2020-06-29 11:30:16.204 INFO INPUT: Device [Throttle - HOTAS Warthog (42263A40-1795-11ea-800e-444553540000)] created deviceId = -1
2020-06-29 11:30:16.204 INFO INPUT: Joystick created[Throttle - HOTAS Warthog (42263A40-1795-11ea-800e-444553540000)], ForceFeedback: no
2020-06-29 11:30:16.204 INFO INPUT: Device [Mouse] created deviceId = -1
2020-06-29 11:30:16.326 INFO SOUND: Using driver: wasapi
2020-06-29 11:30:16.326 INFO SOUND: Found 4 available audio device(s):
2020-06-29 11:30:16.326 INFO SOUND: 0: ID: "{0.0.0.0.00000000},{332218f2-969b-4b0e-873b-2e60c99676cf}", Name: "ASUS VR200 (2- NVIDIA High Definition Audio)"
2020-06-29 11:30:16.326 INFO SOUND: 1: ID: "{0.0.0.0.00000000},{4c6228a3-34dd-4d2b-ef09-163d90e8eebb}", Name: "BenQ GW2250H (2- NVIDIA High Definition Audio)"
2020-06-29 11:30:16.326 INFO SOUND: 2: ID: "{0.0.0.0.00000000},{672bfc03-1a5e-4c4c-80e4-53e01ea39ddc}", Name: "Speakers (Plantronics Savi 750)"
2020-06-29 11:30:16.326 INFO SOUND: 3: ID: "{0.0.0.0.00000000},{cb4f27ad-6332-4f23-9546-d3536d0b3a4e}", Name: "Speakers (High Definition Audio Device)"
2020-06-29 11:30:21.436 INFO SOUND: Added sound path: Sounds
2020-06-29 11:30:21.457 INFO SOUNDER: Loaded 207 packed sounders.
2020-06-29 11:30:21.457 INFO Dispatcher: InitLow
2020-06-29 11:30:21.854 INFO NET: Login success.
2020-06-29 11:30:22.593 INFO NET: Got auth data.
2020-06-29 11:30:22.609 INFO DCS: Successfully got authorization data.
2020-06-29 11:30:22.619 INFO Dispatcher: Loading installed modules...
2020-06-29 11:30:23.174 INFO SECURITYCONTROL: Loaded mode/tech/wwl/units/data_module.pak.crypt
2020-06-29 11:30:24.362 INFO Scripting: Scanning usermods...
2020-06-29 11:30:25.053 INFO Scripting: WARNING: task 36 is missing
2020-06-29 11:30:25.996 WARNING LOG: 2 duplicate message(s) skipped.
2020-06-29 11:30:25.996 INFO Scripting: ./CoreNode\WWII Units\entry.lua
2020-06-29 11:30:25.998 INFO Scripting: ./CoreNode\WWII Units\Weapons\Weapons.lua
2020-06-29 11:30:25.998 INFO Scripting: ./CoreNode\WWII Units\Views.lua
2020-06-29 11:30:25.998 INFO Scripting: ./CoreNode\WWII Units\FW-190D-9\FW-190D-9.lua
2020-06-29 11:30:25.999 INFO Scripting: ./CoreNode\WWII Units\FW-190A-8\FW-190A-8.lua
  
```

## Where can I find example MOOSE missions?

The idea with this repository is that you look at the examples, play with them, adjust and tweak and learn.

[https://github.com/FlightControl-Master/MOOSE\\_MISSIONS](https://github.com/FlightControl-Master/MOOSE_MISSIONS)

### Where is the documentation for MOOSE classes?

The landing page for documentation is here:

[https://flightcontrol-master.github.io/MOOSE\\_DOCS\\_DEVELOP/Documentation/index.html](https://flightcontrol-master.github.io/MOOSE_DOCS_DEVELOP/Documentation/index.html)

All of this is actually inside the Moose.lua file. You need to read the documentation for the full modules to understand how to tweak what they do. But you do not need to worry about all the CORE of Moose and scripting right now!

### How to ask for help

Join the MOOSE Discord here: <https://discord.gg/Q5GE5Uk>

Check out the Eagle Dynamics (ED) MOOSE thread here:

<https://forums.eagle.ru/showthread.php?t=138043>

MOOSE is a community project and support are community based, please remember when posting a question:

- Before posting anything, read your logs. We will ask what they say.
- Post your mission and your logs along with what you expected to happen and what actually happened. Some things need the miz, the script and the log all together.
- Do not use vague words this stuff is hard to help with! Be specific.
- The less detail you offer, the less chance you can be helped.
- Don't ask people to check your script. That is what a computer is for.
- Don't say it doesn't work. Or is it broken. Say what it actually does.

**Welcome to MOOSE and good luck!**

### Intermediate scripting

Welcome to the start of the Intermediate Scripting section of the complete guide. The intermediate guide is everything. Very few people can skip the contents of the Intermediate level, so it encompasses almost the entire guide. If a chapter has not been marked as "Intermediate" it can most likely be skipped as optional.

There are places where the complexity gets a little intense. Please use this as a reference and do not panic if things get beyond you. Take a break, do something different, enjoy your nightmare and ask the therapist about it the next day. Or Discord.

The shopping list for what you need to set out on beginning to script is a lot longer than the one you had before. This is the journey ahead:

- Bookmark our resources like documentation
- Discover the location of the Moose.lua (dev and released versions)
- Become familiar with researching the MOOSE documentation.
- Setup Eclipse LDT (or VSC if proficient)
- Installed a log tailer to track dcs logs in real time
- Learn about the DCS log and errors
- Learn how to add debugging
- Learn some Troubleshooting processes
- Understand the limits of the SSE by reading Hoggit
- Learn how to run and execute a MOOSE script
- Learn how to execute a script dynamically with 'loadfile'
- Learn Lua 101, basic rules
- Read Lua online Ch1-4 quickly
- Learn about SPAWN (Moose 101) and create your first scripts using this class
- Learn how methods are called
- Start to understand about Object Oriented inheritance and hierarchy
- Learn some of the structures in Moose – core, wrapper, functional etc.
- Learn how to use the Moose Documentation
- Learn the CORE MOOSE classes (201)
- Learn the Remaining CORE classes (301)

### Bookmarks & References (Intermediate)

You need a library of web pages to have at your fingertips. Bookmarking from a browser is the only way to keep URL's together. A good Browser like Chrome or Firefox can synchronise your Bookmarks across all your devices. Since MOOSE script writing requires a large library of references, bookmarking the ones most visited will save tons of time. This is part of your extended setup for beyond basic scripting and it is the first thing you simply must do.

#### Documentation landing page

The full version of Moose.lua also contains all the documentation

[https://flightcontrol-master.github.io/MOOSE\\_DOCS\\_DEVELOP/Documentation/index.html](https://flightcontrol-master.github.io/MOOSE_DOCS_DEVELOP/Documentation/index.html)

#### MOOSE releases list Github page

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

#### MOOSE.lua development version static file

[https://github.com/FlightControl-Master/MOOSE\\_INCLUDE/tree/develop/MOOSE\\_Include\\_Static](https://github.com/FlightControl-Master/MOOSE_INCLUDE/tree/develop/MOOSE_Include_Static)

#### Demo missions repository

These illustrate the utility of many MOOSE functions

[https://github.com/FlightControl-Master/MOOSE\\_MISSIONS](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 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/Simulator_Scripting_Engine_Documentation)  
[https://wiki.hoggitworld.com/view/Category:Singleton\\_Functions](https://wiki.hoggitworld.com/view/Category:Singleton_Functions)  
[https://wiki.hoggitworld.com/view/Category:Class\\_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://www.lua.org/manual/5.1/>

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

### Where Do I Find MOOSE Scripts?

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

[https://github.com/FlightControl-Master/MOOSE\\_MISSIONS](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>

### 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](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](https://github.com/FlightControl-Master/MOOSE_DOCS/tree/master/Documentation)

### Checkpoint reached:

- Bookmark our resources like documentation
- Discover the location of the Moose.lua (dev and released versions)
- Become familiar with researching the MOOSE documentation.

### Configuring an IDE like LDT / VSC (Intermediate)

Why do you need Lua Development Tools? Well, to write a script you could use a small tool like Notepad ++, but LDT gives you a lot more features:

- It does Syntax checking; so if you didn't close an if ... then loop, or missed a comment, or wrote something wrong, it tells you.
- It can be plugged into MOOSE and learn all the MOOSE commands and then tell you what they do!
- It can use that MOOSE project to also do autocomplete. This is called "Intellisense" and is a huge time saver.
- It can be connected to a debugger to give additional debugging information on you script.

To set up LDT, follow this guide **very** carefully:

<https://youtu.be/C5yKu1BGpVQ>

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.

<https://forums.eagle.ru/showpost.php?p=4244387&postcount=3>

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

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

IntelliJ 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.

#### Checkpoint reached:

- Setup Eclipse LDT (or VSC if proficient)



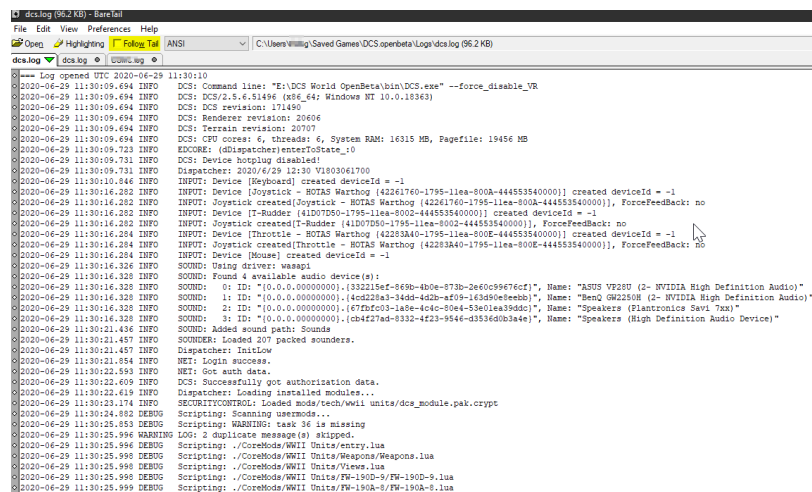
## Logs, Troubleshooting & Debugging (Intermediate)

All script errors end up in the DCS.log. A massive proportion of questions on Discord are answered by looking at the logs. At first, the error appears bewildering and gibberish, but actually they are all straightforward.

Your Saved Games folder has a folder with the name of your install. It can be "DCS" or "DCS.openbeta" or something else, so beware if you have messed about with folder names in the past. C:\Users\<USERNAME>\Saved Games\DCS.openbeta\Logs\**dcs.log**

## Log tailing

<https://www.baremetalsoft.com/baretail/> Baretail is a tiny free program will update in real time as your mission progresses; so not only can you follow your logs as they happen, but they highlight errors which can be very helpful in understanding where you messed up. Glogg also does the same thing and you can get it here <https://glogg.bonnefon.org/> You can also tail in Notepad++. We only recommend a tailer because actually you get to learn a lot about DCS just having a glance at the logs. Watching the script progress in realtime enables you to see the exact moment your script messed up, which is usually right away. And imagine if you weren't watching the logs, you stopped and went to Discord and said, "Help, my script isn't working". No one would answer because you've provided no information (and people can tell you didn't read anything) you will become stuck and frustrated, just because of a typo you missed.



Script troubleshooting and debugging are required skills during code creation; 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

### The DCS Log and Errors

The DCS.log, the primary script debugging tool, resides in: \Saved Games\DCS\Logs\dcsl.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:

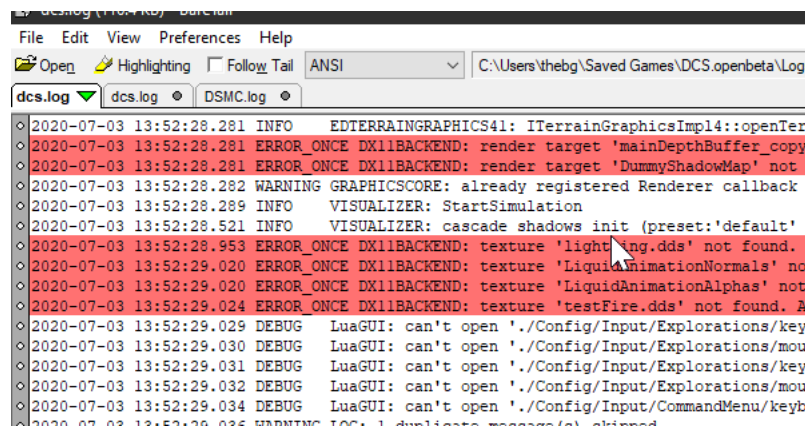
```
ERROR Scripting <we goofed here>
```

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](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.



Red highlight for "ERROR"

## Debugging

When a script misbehaves for an unknown reason, add extra lines of script code that output additional information into the logs. This can be as simple as having logic loops detailed with a few log entries to show you which way the script decided to go:

```
if something1 then
    env.info("something1 is true")
else
    env.info("something2 is true")
end
```

Serializing mitigates the limitation that only text can be output by making an unreadable datatype into a readable string. In other words, you cannot write a *table* to text in something like a message, because a message requires the string datatype. 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:

```
function s(obj) -- obj is the data parameter you want to deserialise
    local Result = routines.utils.oneLineSerialize(obj)
    MESSAGE:New(Result,10):ToAll()
    env.info(Result)
    return Result
end
```

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 Lua. 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 'loggesse'. Eliminating the error involves reading the logs with a log tailer, interpreting the 'loggesse', 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)

### Checkpoint reached:

- Installed a log tailer to track dcs logs in real time
- Learn about the DCS log and errors
- Learn how to add debugging
- Learn some Troubleshooting processes

## Lua 101 (Intermediate)

### 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 is case sensitive:** 'SPAWN' is not the same as, 'Spawn'.

### Lua Essentials - Datatypes

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:

|          |       |       |        |
|----------|-------|-------|--------|
| and      | break | do    | else   |
| elseif   | end   | false | for    |
| function | if    | in    | local  |
| nil      | not   | or    | repeat |
| return   | then  | true  | until  |
| while    |       |       |        |

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

Functions are what make programs output 'something'. The Lua language, the SSE and MOOSE all contain some pre-defined functions. 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 "something" 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.

### Booleans

A datatype of Boolean is either true or false <https://www.lua.org/pil/2.2.html>

### Strings.

Lua strings are a data type consisting of text in either single or 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>

When evaluating two things you can use the signs above; less than, greater than, less than or equal to... etc. 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.html>

'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 from top to bottom once per occasion the script is called. Whilst scripts can have loops, schedules and triggers firing, generally they complete running in a millisecond or so. DCS can have any amount of scripts run without limit. You can run separate scripts or put them together in one big super script. Once the script has executed it will live in memory and if there are no more triggers or timers, it will likely do nothing other than what it did at the instant it was run.

Lua stores variables and functions in memory for use when called by the script, and the script will error 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.

### 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 entries 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.

### Now read Ch1-4

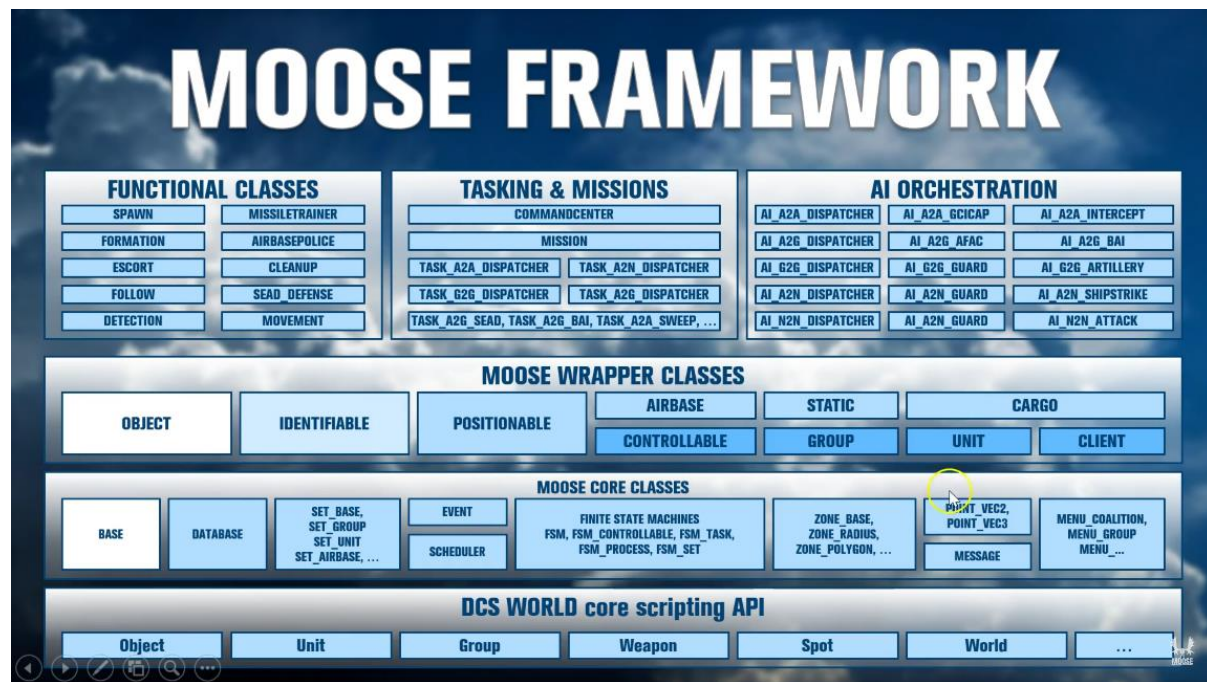
Learning a scripting language without writing a thing requires a leap of faith and is entirely abstract. However this is a prerequisite you just need to skim. You will come back frequently. We would suggest that you skim through the actual Lua online manual, chapters 1-4 here. <https://www.lua.org/pil/1.html> It has a lot of long words and isn't written for fun. But it is relevant, massively relevant. It's a building block learning fundament.

### Checkpoint reached:

- Read Lua online Ch1-4 quickly
- Learn Lua 101, basic rules

## The MOOSE Framework Overview

**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.

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://youtu.be/ZqvdUFhKX4o?t=618> (recommend 1.5x speed).

### The DCS Simulator Scripting Engine (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 behaviours. 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.

It would be a good idea at this point for a new scripter to understand exactly how limited MOOSE really is, but reading the Hoggit wiki:

[https://wiki.hoggitworld.com/view/Simulator\\_Scripting\\_Engine\\_Documentation](https://wiki.hoggitworld.com/view/Simulator_Scripting_Engine_Documentation)  
[https://wiki.hoggitworld.com/view/Category:Singleton\\_Functions](https://wiki.hoggitworld.com/view/Category:Singleton_Functions)  
[https://wiki.hoggitworld.com/view/Category:Class\\_Functions](https://wiki.hoggitworld.com/view/Category:Class_Functions)

The point of this is to counter questions like, "Can I spawn a client slot, can I damage a client plane, can I get the tgp loadout from the plane" The thing is, you just can't do so many things and until, as a scripter you understand what you can, **and cannot** do, you will be eternally facing brick walls, confused and frustrated.

### Running Scripts in DCS

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.

We are mostly concerned with delivering scripts from the main triggers, DO SCRIPT or DO SCRIPT FILE. We can run as many as we like, join different blocks and so on, but the one problem we face is that to change a script in DCS, we have to return to the Mission

Editor, open the triggers and re embed the new script. This is a lot of loading time which is why using a faster workflow is absolutely critical to your success.

### Running scripts directly from Hard Drive

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.

This method also benefits from having errors in that script read out with accurate line numbers, because embedded scripts are decompressed into the Temp directory and given temporary names.

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. **[A good, fast workflow is explained in detail in this video:](#)**

<https://youtu.be/BMKBXjjKiDI>

If you set this workflow up this way, you are likely to succeed in learning to script because it will be a faster learning experience. A good workflow is vital.

### Checkpoint reached:

- Understand the limits of the SSE by reading Hoggit
- Learn how to run and execute a MOOSE script
- Learn how to execute a script dynamically with 'loadfile'

## MOOSE 101 – SPAWN (Intermediate)

Experience tells us the best introduction for learning MOOSE is seeing how the most common MOOSE function works and getting instant gratification. So, let us 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.

## Anatomy of a 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" )
```

**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(SpawnTemplatePrefix)
```

which can be 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 Messenger, periodically runs the included function()

Messenger = 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

        Messenger: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
```

### 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 they research the MOOSE resources to determine how the idea might be realized in a mission and attempt 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:

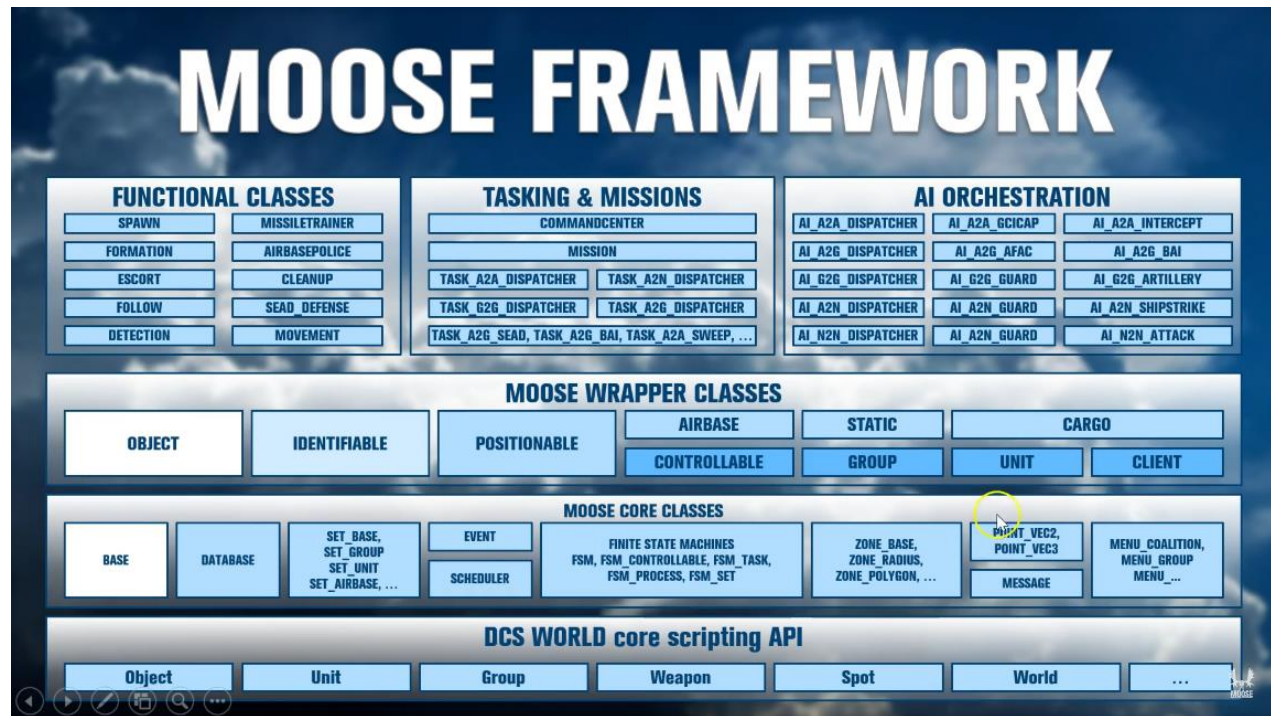
- OnSpawnGroup()
- Randomization
- Multiple spawns from an array in a "for loop"
- InitLimit() can fail because one neglected to read the arguments or doesn't know that UNIT and GROUP classes have different functions.
- Spawn() returns the group name
- Using ZONEs and tables

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.

### Checkpoint reached

- Learn about SPAWN (Moose 101) and create your first scripts using this class

## MOOSE Documentation and Object Oriented structure (Intermediate)



### Inheritance in Object Oriented Programming

**Inheritance** is a mechanism in which one class acquires the property of another class. For example, a child **inherits** the traits of his/her parents. With **inheritance**, we can reuse the fields and methods of the existing class. Hence, **inheritance** facilitates Reusability and is an important concept of Object Oriented Programming. 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.

What does that actually **mean**, in practice!?

In the SSE, to issue a command or option to a group, you find the group, find its controller, send the controller a message to set the option.

```
local _grp = getGroup("Green State")
```

```
local _controller = _grp:getController()  
Controller.setOption(_controller, AI.Option.Ground.id.ALARM_STATE,  
AI.Option.Ground.val.ALARM_STATE.AUTO)
```

In MOOSE, because GROUP inherits CONTROLLABLE, you can issue the command to the GROUP *directly*, without having to get the controller of the Group.

```
GreenStateGroup = GROUP:FindByName( "Green State" )  
GreenStateGroup:OptionAlarmStateGreen()
```

So here we see directly an advantage of OOP and inheritance in simplification. This is just one example, but there are many more.

The documentation of MOOSE requires some explanation. You notice the dot notation continuing in the docs, e.g. Core.Zone. This means Zone is a Core Class.

### What is a Core Class

CORE classes are the bottom level fundamental blocks MOOSE. They include the Database, Zone, Point, Menu, Message, FSM, Scheduler, Event, SET.

- Database: MOOSE maintains a Database of all the mission objects and keeps this updated throughout the mission.
- Zone: Everything to do with classic zones and polygon and moving zones.
- Point: Vector2, Vector3 and game coordinates and positions.
- Menu: Constructing F10 menus
- Message: Constructing messages to screen
- FSM: Finite State Machine is a class for handling and creating FSM processes  
[https://en.wikipedia.org/wiki/Event-driven\\_finite-state\\_machine](https://en.wikipedia.org/wiki/Event-driven_finite-state_machine)
- Event: create Event Handlers for retrieving data from in game events fired by the DCS SSE.
- Scheduler: This class handles and simplifies usage of the timer functions in DCS
- SET: Sets are automated tables of groups or units or other things that will be maintained by MOOSE.

### What is an Object?

In this case an object is literally a table, depending on what the object is, it might have its name, its ID, and some other details. To see the contents, go to the debugging section in the Advanced Troubleshooting chapter and use the provided function to serialise a table to text and have a look. So, when we say the Group Object, this is a reference to the Group in DCS. Not actually its name, but the collection of its name, its units, its ID, where it is and so on.

### What is a Wrapper?

A wrapper is another word for the Object. But for MOOSE the object is wrapped in a more Moosey way. Wrapper classes contain group, unit, airbase, static, client, cargo, controllable, positionable. The wrapper classes tend to focus on "things" in DCS.

### Functional, Tasking, AI, Ops, Utils etc

Functional and Ops classes “do” things, like Spawn. AI classes handle... well AI. Utils is a class devoted to all the tools for conversions etc. Tasking is a class devoted to setting messages to the player for a Task they can join.

### How to read the Documentation

Documentation on reading Documentation!? This needs to be explained because there is more than meets the eye of the casual first time observer. Moose Documentation is generated from the code itself. All the docs are inside moose.lua. The Documentation home page [https://flightcontrol-master.github.io/MOOSE\\_DOCS\\_DEVELOP/Documentation/index.html](https://flightcontrol-master.github.io/MOOSE_DOCS_DEVELOP/Documentation/index.html) looks like this:



#### **Core.UserSound**

**Core** - Manage user sound.



#### **Core.Velocity**

**Core** - Models a velocity or speed, which can be expressed in various formats according the settings.



#### **Core.Zone**

**Core** - Define zones within your mission of various forms, with various capabilities.



#### **DCS**

**DCS API** Prototypes

See the [Simulator Scripting Engine Documentation](#) on Hoggit for further explanation and examples.



#### **ENUMS**

**Utilities** Enumerators.




With no Index bar the simplest way of searching for a class will be using the page search CTRL+F, to look for `Wrapper.Group`. Let's go to Group and see how the page is formatted:

The first part is devoted to an explanation of what the class actually does or is. If the Documentation says something like: "IMPORTANT", then it really is. The explanation here tells the reader that you cannot create a New group... there is no GROUP:New() method and the way you get a group is via Find() method.

After the preamble, comes a long list of Methods that belong to the GROUP class. Because these are GROUP class methods, they only work with a GROUP object wrapper. The exception is inherited classes and their methods. An inherited class like CONTORLLABLE has methods that work on a GROUP Class. It's confusing, but this is a fundamental learning point to how to use methods and what objects they work on.

# WRAPPER - GROUP




---

## Module Wrapper.Group

**Wrapper** -- GROUP wraps the DCS Class Group objects.

The [GROUP](#) class is a wrapper class to handle the DCS Group objects.

**Features:**

- Support all DCS Group APIs.
- Enhance with Group specific APIs not in the DCS Group API set.
- Handle local Group Controller.
- Manage the "state" of the DCS Group.

**IMPORTANT: ONE SHOULD NEVER SANATIZE these GROUP OBJECT REFERENCES! (make the GROUP object references nil).**

For each DCS Group object alive within a running mission, a GROUP wrapper object (instance) will be created within the [DATABASE](#) object. This is done at the beginning of the mission (when the mission starts), and dynamically when new DCS Group objects are spawned (using the [SPAWN](#) class).

The GROUP class does not contain a :New() method, rather it provides :Find() methods to retrieve the object reference using the DCS Group or the DCS GroupName.

The GROUP methods will reference the DCS Group object by name when it is needed during API execution. If the DCS Group object does not exist or is nil, the GROUP methods will return nil and may log an exception in the DCS.log file.

**Author: FlightControl**

**Contributions:**

- **Entropy, Aflnegan:** Came up with the requirement for AIOnOff().

---

**Global(s)**

---

**Global GROUP**

Wrapper class of the DCS world Group object.

This list is prefaced with all the inherited classes. So when you see "Group extends controllabile, extends positionabile" etc....this is explaining the inheritance of the class.

The list of methods starts with summary information. You get the `CLASS:Method()` and a description of what it does. These are all hyperlinked to the exact Method where you can find more detail.

| Type GROUP  |   |
|---|---|
| GROUP, extends Wrapper.Controllable#CONTROLLABLE, extends Wrapper.Positionable#POSITIONABLE, extends Wrapper.Identifiable#IDENTIFIABLE, extends Wrapper.Object#OBJECT, extends Core.Base#BASE |   |
| Fields and Methods inherited from GROUP   | Description   |
| GROUP:Activate(delay)   | Activates a late activated GROUP.                   |
| GROUP:AllOnGround()   | Returns if all units of the group are on the ground |
| GROUP.Attribute   |   |
| GROUP:CalculateThreatLevelA2G()   | Calculate the maximum A2G threat level of the Group |
| GROUP:CopyRoute(Begin, End, Randomize, Radius)  | Return the route of a group by using the Core.Data  |
| GROUP:CountAliveUnits()   | Count number of alive units in the group.           |



Finally, if you follow the link of the summary method you will come to the methods detailed description. This may require some explaining if you are unfamiliar with how functions are explained in documentation.

### GROUP:HasAttribute(attribute, all)

Check if at least one (or all) unit(s) has (have) a certain attribute.

See [hoggit documentation](#).

#### Defined in:

GROUP

#### Parameters:

#string **attribute**

The name of the attribute the group is supposed to have. Valid attributes can be found in the "db\_attributes.lua" file which is located at in "C:\Program Files\Eagle Dynamics\DCS World\Scripts\Database".

#boolean **all**

If true, all units of the group must have the attribute in order to return true. Default is only one unit of a heterogenous group needs to have the attribute.

#### Return value:

#boolean:

Group has this attribute.

The Parameters will list, in order, the expected things you must type into the method to make it not error. In the example above, the 1<sup>st</sup> parameter of GROUP:HasAttribute() is "attribute". "Attribute" must be of the datatype "string". The important clue here is the datatype is put right next to the parameter. Valid datatypes are normal Lua datatypes like:

- String ("text")
- Number (12)
- Boolean (true or false)
- Table (a table {} )
- nil and functions are also datatypes that can be used as Parameters. Often a function is written to accept nil.

## Optional Parameters

One thing to look out for is if the word **(optional)**

### GROUP:CommandDeactivateICLS(Delay)

Deactivate the ICLS of the CONTROLLABLE.

#### Defined in:

[Wrapper.Controllable#CONTROLLABLE](#)

#### Parameter:

#number **Delay**

**(Optional)** Delay in seconds before the ICLS is deactivated.

#### Return value:

[#CONTROLLABLE:](#)

self

This means that you do not need to enter a parameter at all for the method to work on the GROUP. This is because the method most likely has a default. The notes will make that clearer.

The last section shows you the Return value, or what you get back if you use the function correctly. It will also explain the datatype to expect.

Return = GROUPobj:Method(Parameter1, Parameter2)

“self” is a common return when nothing else might be applicable – i.e. the method is doing something like making a tank shoot, that is the result of the method, but you know you sent it correctly if the return value was checked and was something that the documentation said it should be.

The final thing to note on Documentation pages is that if there are inherited classes that you can use the Method on, they are listed under the main class. This will give the appearance that the documentation repeats itself a lot, but since inheritance is such a major part of OOP, the documentation of inherited classes is significant!

### Checkpoint reached

- Learn how methods are called
- Start to understand about Object Oriented inheritance and hierarchy
- Learn some of the structures in Moose – core, wrapper, functional etc.
- Learn how to use the Moose Documentation

## MOOSE Scripting 201 (Intermediate)

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

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\(SchedulerObject, SchedulerFunction, SchedulerArguments, Start, Repeat, RandomizeFactor, Stop\)](#)

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 health
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.

## **MOOSE User Guide**

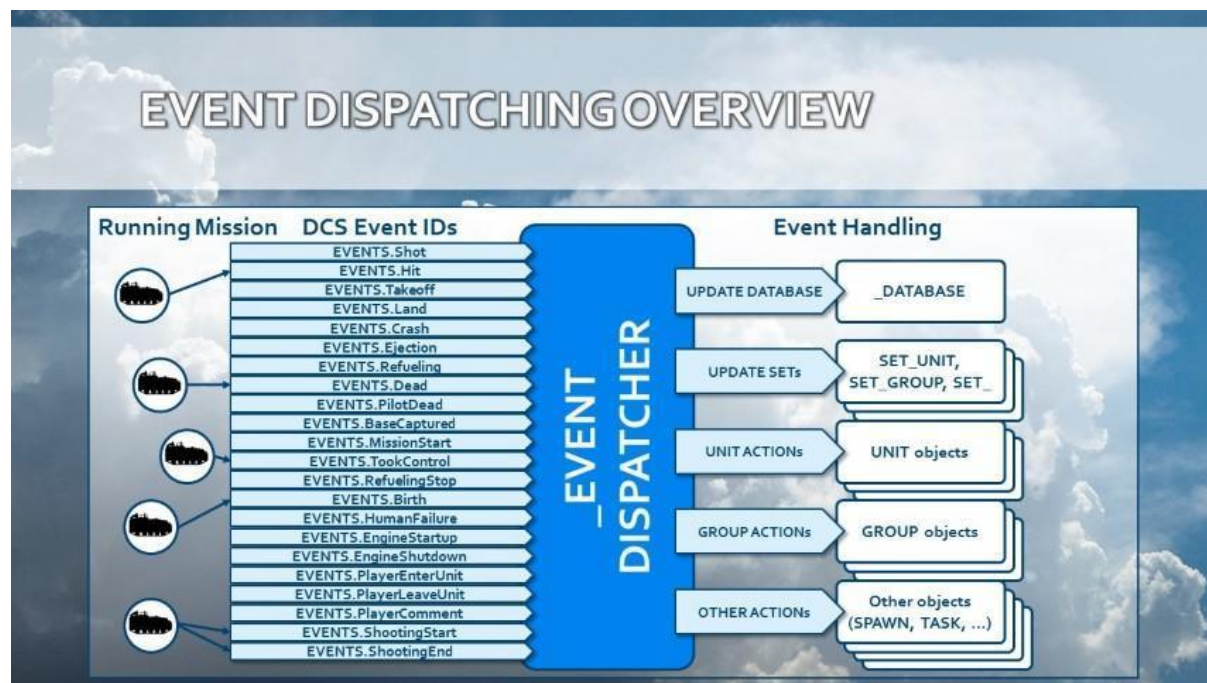
A couple of weeks should prove sufficient to become comfortable with these classes. The ability to create a series of mission actions using these methods indicates a good grasp of the basics.

## MOOSE Scripting 301 (Intermediate)

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

Events are realtime feedback events form the game. 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

The CONTROLLABLE class is almost redundant to the GROUP and UNIT wrappers. An AI Controller for a Unit or Group is the interface to controlling that group or unit. This is where you send commands. A good description of how these commands work is this Hoggit link [https://wiki.hoggitworld.com/view/Mission\\_Editor:\\_AI\\_Tasking](https://wiki.hoggitworld.com/view/Mission_Editor:_AI_Tasking). In MOOSE, the Unit or Group inherits the controller so you never need to define the controller as a separate thing. In pure SSE lua scripting you have to find the group, then getController() on it before issuing the command.

### 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 are well documented, require only a few lines of code and thus do not appreciably advance one's scripting competence. 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:

## MOOSE User Guide

- 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.

### Checkpoint reached

- Learn the CORE MOOSE classes (201)
- Learn the Remaining CORE classes (301)

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

### Advanced MOOSE

The tallest hurdle is getting started. There is not much else other than working through individual scripting problems or maths or design of scripts. You won't ever feel that confident, but you get faster at scripting and the things you improve on, are learning what you can and cannot do with the SSE, as well as how easy it is to break DCS.

The next step is quite optional.

You can:

- Develop for Moose
- Setup the Lua Debugger
- Contribute to helping
- Go outside of the Mission environment and explore

Interactive debugger guide: [https://flightcontrol-master.github.io/MOOSE\\_DOCS/Interactive\\_Debug\\_Guide.html](https://flightcontrol-master.github.io/MOOSE_DOCS/Interactive_Debug_Guide.html)

Contributing to MOOSE (deprecated)  
[https://flightcontrol-master.github.io/MOOSE\\_DOCS/Contribution\\_Guide.html](https://flightcontrol-master.github.io/MOOSE_DOCS/Contribution_Guide.html)

To build repositories of your own and build Moose from the dynamic loader, follow this guide  
[https://flightcontrol-master.github.io/MOOSE\\_DOCS\\_DEVELOP/Beta\\_Test\\_Guide.html](https://flightcontrol-master.github.io/MOOSE_DOCS_DEVELOP/Beta_Test_Guide.html)

You require to download Github desktop.

To contribute, please contact any of the team on Discord. You will need to have your Github username handy. Helping is more of a mindset. Anyone can help in a multitude of ways. Writing classes for Moose is a matter of having your branch approve to be merged and waving your hand to tell someone!

## Appendix A - How DCS Works

### Evolution of DCS SSE

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 millennials, who rely heavily on YouTube videos and Google search, experienced people remain gold mines of information.

DCS has multiple Lua environments, each totally isolated from the others and having their own distinct **scope**. The Lua environment of interest to MOOSE scripters is the "Mission" environment. The other key Lua environment is the "server" 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. MOOSE is not meant for server side execution. Any modifications, however, need a full restart of the game engine; so, working in the Net environment proves problematic.

### 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.

| Name       | Size    | Pack... | Moc |
|------------|---------|---------|-----|
| VHF_RADIO  | 140     | 87      |     |
| Scripts    | 34 197  | 4 682   |     |
| I10n       | 2 54... | 427 ... |     |
| Config     | 111 ... | 8 150   |     |
| warehouses | 27 762  | 672     |     |
| options    | 8 603   | 1 993   |     |
| mission    | 300 ... | 16 937  |     |

**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; \I10n\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>

```

1 mission =
2 {
3   ["requiredModules"] =
4   {
5     ["TF-51D Mustang by Eagle Dynamics"] = "TF-51D Mustang by Eagle Dynamics",
6     }, -- end of ["requiredModules"]
7   ["date"] =
8   {
9     ["Year"] = 2011,
10    ["Day"] = 1,
11    ["Month"] = 6,
12    }, -- end of ["date"]
13  ["trig"] =
14  {
15    ["actions"] =
16    {
17      [1] = "a_do_script_file(getValueResourceByKey(\"ResKey_Action_323\"));",
18      [2] = "a_do_script(getValueDictByKey(\"DictKey_ActionText_627\")); mission.trig.func[2]=nil;",
19      }, -- end of ["actions"]
20    ["events"] =
21    {
22      }, -- end of ["events"]
23    ["custom"] =
24    {
25      }, -- end of ["custom"]
26    ["func"] =
27    {
28      [2] = "if mission.trig.conditions[2]() then mission.trig.actions[2]() end",
29      }, -- end of ["func"]
30    ["flag"] =
31    {
32      [1] = true,
33      [2] = true,
34      }, -- end of ["flag"]
35    ["conditions"] =
36    {
37      [1] = "return(true)",
38      [2] = "return(true)",
39      }, -- end of ["conditions"]
40    ["customStartup"] =
41    {
42      }, -- end of ["customStartup"]
43    ["funcStartup"] =
44    {
45      [1] = "if mission.trig.conditions[1]() then mission.trig.actions[1]() end",
46      }, -- end of ["funcStartup"]
47    }, -- end of ["trig"]
48  ["result"] =
49  {
50    }
51  }
52 }
```

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 \10n\DEFAULT.

**File; \Config\View** folder contains View.lua, SnapViewsDefault.lua files, which do 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.

```
[ "helicopter" ] =
{
  [ "group" ] =
  {
    [1] =
    {
      [ "lateActivation" ] = true,
      [ "tasks" ] =
      {
        -- end of [ "tasks" ]
        [ "radioSet" ] = false,
        [ "task" ] = "CAS",
        [ "uncontrolled" ] = false,
        [ "route" ] =
        {
          [ "routeRelativeTOT" ] = true,
          [ "points" ] =
          {
            [1] =
            {
              [ "alt" ] = 3353,
              [ "action" ] = "Turning Point",
              [ "alt_type" ] = "BARO",
              [ "speed" ] = 55.555555555556,
              [ "task" ] =
              {
                [ "id" ] = "ComboTask",
                [ "params" ] =
                {
                  [ "tasks" ] =
                  {
                    [1] =
                    {
                      [ "number" ] = 1,
                      [ "key" ] = "CAS",
                      [ "id" ] = "EngageTargets",
                      [ "enabled" ] = true,
                      [ "auto" ] = true,
                      [ "params" ] =
                      {
                        [ "targetTypes" ] =
                        {
                          [1] = "Helicopters",
                          [2] = "Ground Units",
                          [3] = "Light armed ships",
                        },
                        -- end of [ "targetTypes" ]
                        [ "priority" ] = 0,
                      },
                      -- end of [ "params" ]
                    },
                    -- end of [1]
                  },
                  -- end of [ "tasks" ]
                },
                -- end of [ "params" ]
              },
              -- end of [ "task" ]
              [ "type" ] = "Turning Point",
              [ "ETA" ] = 0,
              [ "ETA_locked" ] = true,
              [ "y" ] = 731925.66285714,
              [ "x" ] = -166076.33428572,
            },
          },
        },
      },
    },
  },
}
```



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.

```
[ "units" ] =
{
  [1] =
  {
    [ "type" ] = "Challenger2",
    [ "transportable" ] =
    {
      [ "randomTransportable" ] = false,
    }, -- end of [ "transportable" ]
    [ "unitId" ] = 130,
    [ "skill" ] = "Average",
    [ "y" ] = 688333.91531068,
    [ "x" ] = -282738.75757671,
    [ "name" ] = "DictKey_UnitName_258",
    [ "heading" ] = -1.9526961075565,
    [ "playerCanDrive" ] = false,
  }, -- end of [1]
```

### 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.

### Appendix B - 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 2020, MOOSE had a large and busy Discord server with over 1397 members, more than doubling year on year, huge documentation and video libraries, several code contributors/champions, and currently sits at over 147 thousand lines of code and embedded documentation. The 265 page main thread on the ED forums has over 435,000 views. More DCS mission developers are discovering the wonderful world of MOOSE every day.

## Appendix C - Troubleshooting techniques

*"When you have eliminated all which is impossible, then whatever remains, however improbable, must be the truth."* —Sherlock Holmes

**T/S:** Shorthand for trouble shooting, the act of working out the cause and fix of an issue

**Symptoms:** Symptoms are apparent *displays* of the presence of a specific problem. They are not the problem itself. There is the problem, and the problems symptoms.

**Probable cause:** The nominal direct line reason for the issue.

**Root cause:** The original start of the trail of causes leading to the final symptom.

**RCA:** Root Cause Analysis – The process of chasing the data and of [5 whys](#) to understand the original start of the problem.

**Intermittent issues:** Non reproduceable issues, issues that do not occur when the same set of data is passed as an input. Intermittency is the T/S's nemesis.

**Timing issues:** Timing issues are a specific type of intermittent issue that relies on something called a **race-condition**. Race conditions are when two or more events that are 'time based' may complete in a different order causing something unexpected. A good example of issues that are caused by race conditions are testing for GROUP's that are alive just after they are spawned. Example: The code execution moves from the Spawning of a Group and continues. The Lua responsible for Spawning the Group performs the lower level C code to the simulation, makes prechecks (surface, validity) and brings the groups model into the game. The Group's arrival triggers a birth event. The birth event is handled by Moose database. Moose adds the Group to the Database. If a `GROUP:FindByName()` occurs on that group before it exists in the world, even though the Spawn was called first, then FindByName will return nil as the object doesn't exist (at the time) Also happens with destroyed groups, the lag between the group being removed from the database and the act of it "dying".

### Simplification/Isolation

When a MOOSE script produces unexpected results, remove 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. Isolation is a form of Bracketing; however it is following the principal of simplifying the issue to the barest form to remove any complexity. Reproduce in the simplest of forms and you have less complexity to muddle through.

### Reproduction

Reproducing the problem is the second step to effective troubleshooting, but many people do not know why it is required. When you can cause the problem on demand, you know the steps to make something happen and have "captured" the issue. You can then institute one of the common T/S techniques like bracketing and change the inputs. Providing the miz file is part of the steps to reproduce. Reproduction should be done locally on the tester's workstation. Without reproduction, the problem is not transportable. If the problem is not transportable, the Root cause could still be something specific to the

local environment or process. E.g. a person has a broken script. They provide the mission to someone to check. The mission runs fine (WFM – works for me). This shows the problem was either a) not isolated effectively b) not reproduced effectively. c) a problem of environment (scripts in the profile, installation issues, PEBKAC)

### Bracketing/BlackBox

Bracketing or Black Box troubleshooting is simply ruling things out until you are left with the only possibility. Anyone can do Black Box troubleshooting because you have no requirement to know anything under the hood. You just take the inputs and rearrange them so that on each execution of the reproduction steps you rule out a possible cause. The opposite, White Box T/S, assumes you understand all the code and processes absolutely. In most cases this is never available. DCS will forever be a Black Box scenario! E.g. My table has 18 groups. The script intermittently errors as it goes through the table. One of the objects is the cause. Bracketing would run the script with objects 1-9, see if the problem reproduces. If so, we have eliminated objects 9-18 and we can use objects 1-8 to test against. The same technique works by removing processes from a multi process operation or segments or any other large amount of inputs.

### Be Data Driven

Forming conclusions about an issue should never be made on a gut instinct, even a small conclusion can close off an avenue of investigation. All avenues remain open until the data presented shows that it is not. Chasing the data means to follow where the evidence (data) points to, and specifically you can use this to see what executed, to validate each step of the way. Debugging comments is a way of being Data Driven. You put in a debug comment and see it executes and you know that the code processed this normally.

## Appendix D - 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('K0Static1')) < 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 preferable)  
  
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 coalition 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**

-----

## MOOSE User Guide

```
-- 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
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
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)
```

## MOOSE User Guide

```
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
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
end
return table.concat(t_str)
else
    return ""
end
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
end
end
end
end

```

## MOOSE User Guide

```
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
```



## MOOSE User Guide

```
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)`