# Troubleshooting & self help with issues

Script troubleshooting and debugging comprises **the single most important skill for improving Lua competency**. The DCS logs comprise the primary tool to understand 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.

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

* Using Discord rather than Forums
* Using the PC app rather than your phone
* Asking good questions and being specific
* Using Discord markup and correct pasting formats
* Having a log tailer with highlighting and watching your logs in real time <https://www.baremetalsoft.com/baretail/> will do.
* 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
* Serializing a table to discover its contents

How to succeed in asking for help

Remembering our manners, successfully getting help on Discord or the Forums depends on two things.

1. The tone it was asked in
2. Whether it is answerable or not

Someone once said, “There are no stupid questions, only stupid people”. This is a play on words and fundamentally is complete garbage. We all know what a stupid question is. However, let’s deal with ‘poorly put’ questions that are unlikely to get help. Assuming you asked nicely, the second part requires enough accompanying information in order to answer the question.

**You never saw someone put up their hand in class to a question they couldn’t answer, right?**

Neither will anyone else volunteer an answer, to a question they cannot answer. Let’s look at REAL examples:

**“Can anyone help me with a problem?”** This question is unanswerable, we simply don’t know until the question is put. Additionally, it invites ridicule for the 500 people online thinking up funny replies. I personally ignore these questions as it invites conversation that is obvious and can be skipped. We might help, but it sounds like you are hiding something, and we don’t want to be trapped by offering then to find we know nothing of the topic matter and wasting our time. Ask the question you mean to ask, don’t hang around waiting for someone to say yes, followed by, ‘I don’t know’. The person with the answer may be sleeping thousands of miles away, were you going to wait?

**“Moose is broken”** or **“SPAWN doesn’t work?” “Is SPAWN broken/working?”**

Are all saying, “I’ve goofed somewhere but I am convinced it cannot be me.” Start with the assumption that you goofed and things will run more smoothly. No one is perfect. Ignore what your mother said.

**“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 scripting skills do not include mind reading. The first analytical question is “what script?”, followed by “broken how?” Speed up the process and provide the information so we can skip the obvious follow up questions.

**“Why doesn’t this script work?” “Can you see anything wrong with this script”, “Can you check my script” @here Anybody know why the C-130 group isn't spawning? Must be something wrong I can't see**

[lists script snippet] (and nothing else)

I usually ignore these. To this style of question, a reviewer might respond with “What did DCS.log say*?”* because we are not computers. Scripts are made to be read by computers, not humans who use native languages. What is really an issue here is that you decided to put a script on Discord without checking the logs and demonstrated you don’t know the value of reading your own logs. So the correct answer should be, “*Get a log reader*”.

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]

```

Stop screenshotting code. Even worse… a mobile phone of your screen… Discord allows us to copy and paste and format, and there is a PC application that works better than the web or mobile one. *You cannot use a unique photograph as a search criteria!*

**“I want to do X. I have a great idea but I am unable to script?”**

We all have great ideas, that is why we learned to script. Whilst you might have a great idea, our ideas are more fun to **us**, and no one wants to give up their dreams and hard work for your dreams, with no hard work. If you want to suggest a feature request, write it in detail on GitHub. 9/10 it’s been done before, highly specific and not reuseable, can’t be done, or sounds like drivel.

Follow ups that reply with: **“I can’t supply logs because I am at work”, “I’ll try when I get home”**

Don’t ask questions if you cannot deal with the follow up. Discord is realtime, if you ask a question expect one back. We will ask for logs. And maybe a miz, without mods and your script. Don’t be surprised by us asking questions!

**“Can you take a look at my mission?”**

No. if we want to, we’ll ask. Tell us the problem, the logs, the script etc. This is tantamount to saying that you cannot be bothered to put in any work and thus we can’t be bothered either.

**PMing anyone online at the top of the member list**

This is really bad form. What happens is people start to switch off their visibility in order not to be pestered. There is literally ZERO sensible reason to pick an individual out and nail them with a question. Questions on Discord are there in public view to help others. By removing the option for anyone else to learn from it, you made the answer a very justifiable, *“No, I am a community helper. I do this for everyone and it’s not a private service. If you put your question on Discord I will read it and if I can answer it, I will”*

So the minimum requirement for a good request for help is;

* Being available for follow up questions
* The script itself, preferably whole, especially when you have functions off the snippet that are questionable.
* You read the logs and posted them or a relevant snippet
* The Miz may be requested as some problems require a look at the mission because it may need a reference to a zone or template group name, or the script triggers.
* The expected behaviour and the actual behaviour without words like *broken*, *not working* or other vagueness
* Any special steps to reproduce

Dynamic loading of scripts

Scripts that run embedded as a file are sent to your temporary folder during mission execution and garbled a bit. It means the line number will be hard to work out if you are staring at ten lines as they get jumbled in the temp file.

Also, editing and reloading that script is a bit painful as you must return to the Mission Editor to change it. The preferred workflow for testing is to run your scripts directly from disk by using a DO SCRIPT action such as

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

if you couple this with a trigger that fires from a menu item you will speed up your script fixing and get better log line information. A full guide is here:

* <https://www.youtube.com/watch?v=BMKBXjjKiDI>

The DCS Log & finding Errors with a tailer

The DCS.log, the primary script debugging tool, resides in **\Saved Games\DCS\Logs\dcs.log.** DCS writes everything to this log as it loads, and as the mission 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

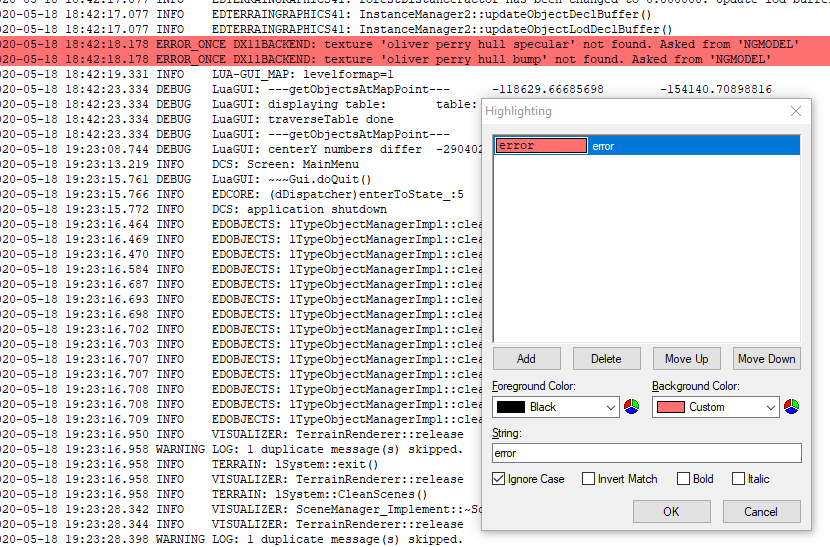
DCS World will create env.info (an INFO log), env.warning (a WARNING log) and env.error (an ERROR log). Each of these log types outputs lines of text to the DCS.log. An env.error() halts the current program execution, whereas env.info does not. The log will print the event **logging level**, the **source** 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. Moose automatically turns off the halt of script processes with errors. The code block will not continue. However if the code that errored is in a different, lower block to the script, the script will continue, and if it calls that bad block again, will throw an error every time it is called. Refer to:

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

A properly configured log tailer highlights errors via a regular expression or ‘regex’; and good practice dictates highlighting env.info’s by putting a keyword in all the error logging when writing the code. This way, a colour code will differentiate the script’s errors from other types.



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

Create a Test Mission

When a MOOSE script produces unexpected results, delete all the code, except for suspect sections and essential functions, and run the stripped-down script in a test mission consisting of only essential mission entities. Different sections of the script can be efficiently tested to isolate the misbehaving code; and the test script can be quickly modified and re-tested until it yields the expected results. It makes a lot of sense to not use mods as if you need to send the mission with mods, chances are it will get sent back with, “I cannot run this, it has mods”

Debugging

**Serializing**

Often you want to know what kind of data is in a variable. Sometimes you cannot simply look inside a variable or table because env.info(MyVariable) contains data other than a string, like a table or a Boolean and you need to know what it is. Lua is very table centric, so being able to display a table such as the return of an Event (The EventData) is good for working out what you can use.

Serializing mitigates the limitation that only text can be output by making an unreadable datatype into a readable string. To discover the contents of a DCS table or something from MOOSE, look inside a SPAWN or ZONE object. Lua has built in serialising functions, and the Mist imported serialising functions are available.

Here is a custom serializing function for printing suspected faulty code to screen, and logs:

functions(obj)

local Result = routines.utils.oneLineSerialize(obj)

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

env.info(Result)

return Result

end

This function can take most objects and return them in a human readable form. It helps when you try to access something and get nil. It also helps you understand code better if you “look inside”.

**Basic logging and comments**

Comment your own code with the double minus “- -". Chances are if you ever look at it again you wont understand it.

Use debugging log outputs like BASE:I or env.info(“text”) to mark your logs with what is happening. You can see the code progress and it will tell you what ran OK. Often I write a env.info(“Code complete”) at the bottom of my scripts so I know it all ran first time.

**Catch all code**

You can also use debugging comments to find out how your script is progressing and where it falls over, this is especially used in conditional statements such as if..then clauses. This is typically called a ‘catch all’, because the code runs through, checking the statements to be true and if not eventually ends up at the bottom “else” clause:

if A:isAlive() then

env.info(“A is alive”)

elseif B:isAlive() then

env.info(“B is alive”)

else

env.info(“nothing is alive”)

end

In that example the last escape clause will catch everything and you know your logic ran to the end of all the conditions and wasn’t caught. You aren’t interested in the 3rd condition of none of the above, in reality, but if it happens that it got there and nothing was evaluated you have to deal with it. Debugging using statements with **if, then, else** is practically everywhere in code and often you put it in during development and take it out when it works. The logs will only say one of the situations in clear text and enable you to understand where things go wrong.

How to read errors

The DCS log, will show the line number the issue began at in the script like so:

MyCode = Rubbish[0] 🡨 Line 147

The logs always record a scripting error, stating the error clock time, script line number and a description of the error. Eliminating the error involves reading the logs with a log tailer, interpreting the error’s odd text, 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 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 because the index was incorrect 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 addon 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)