

cf/x

THE DEBUGGER

Mission Debugger for DCS

STAND-ALONE VERSION

© 2022 - 2024 by Christian Franz and cf/x

Version 2.0.0 – 2024-01-01

PLEASE NOTE

This debugger is a stand-alone, fully functional version of the mission debugger that comes as part of “DML”, a DCS mission toolbox.

If you are using DML you do not need this version; use the one that comes with DML, as it's better maintained, and usually more recent.

Table of Contents

1	Preface	4
2	The Debugger.....	5
2.1	Using The Debugger (Summary)	7
2.1.1	Adding The Debugger to your mission (DML)	7
2.1.2	Adding The Debugger STANDALONE to your mission	7
2.1.3	During design (Mission Editor)	7
2.1.4	In-Mission	8
2.1.5	Debugger commands (overview)	9
2.2	Interactive Use during Missions	18
2.2.1	Remembering it for you wholesale: -help and -?	19
2.2.2	Fundamental commands: -show, -set, -flip and -inc	19
2.2.3	Let the Debugger work for you: -observe and -o	20
2.2.4	Losing interest: -forget	21
2.2.5	Annotating Message History: -note	21
2.2.6	Clearing the Slate: -reset	22
2.2.7	Getting the BIG picture: -snap and -compare.....	22
2.2.8	Bringing in the (big) Guns: -spawn	23
2.2.9	Getting rid of 'em: -remove.....	25
2.2.10	Smoke On! -smoke	26
2.2.11	Boom Baby! - boom	26
2.3	Killer Features.....	27
2.3.1	Q, no longer anon: -q	27
2.3.2	Analyze this: -a	27
2.3.3	W is for WARNING: -w.....	27
2.4	Big Brother: Observers	29
2.4.1	Creating an observer: -new and new for	29
2.4.2	Adding flags to observers: -observe with.....	30
2.4.3	Changing an observer's condition: -update to	30
2.4.4	Supported Observer Conditions	31
2.4.5	Show me what you got: -list	31
2.4.6	Show and tell: -show observename	32
2.4.7	Who's zoomin' who: -who	32
2.4.8	... and Forget Me Nots: -forget with	33

2.4.9	Oh, snap! -snap observername.....	33
2.4.10	The spotless mind: -drop observername	33
2.4.11	Main switch: -start and -stop	34
2.5	Saving the Debugging Log: -save [filename].....	35
2.5.1	Using -save [filename]	35
2.5.2	How to enable -save	36
2.5.3	Do I have to add persistence?	36
2.6	Integration With Mission Editor (Optional)	37
2.6.1	Creating Observers in ME.....	37
2.6.2	Dedicated and Stacked Zones, tracking local flags	38
2.6.3	Activating Event Monitoring.....	38
2.6.4	Setting up Generic Stand-ins	39
2.7	Adding The Debugger to your mission	40
2.7.1	Stand-Alone (NO DML in your mission)	40
2.7.2	DML-enhanced Missions	40
2.7.3	ME Attributes	41
2.8	Bug Hunt - A Live demo	44
2.8.1	Please ignore me: Self Test.....	44
2.8.2	Observing and setting flags to debug your mission	44
2.8.3	Setting flags to skip/advance mission stages	46
2.8.4	Using Observers and snapshots to debug your mission.....	47
2.8.5	Setting up Observers in ME	49
2.8.6	Debugging local flags, flag concurrency.....	50
2.8.7	Now hear this: debugMsg and sayWhen observer attributes.....	52
2.8.8	Mission Discussion: What DML does in this demo	53
2.9	Debug events and More (.miz).....	55
2.9.1	Starting the mission	55
2.9.2	In-Mission	55

1 Preface

This is the *stand-alone* version of “THE DEBUGGER”, which itself is part of the DML Mission Library for DCS. You do not need any DML knowledge to use this debugger in your missions, and you do not have to add DML modules/scripts to your mission in order to use it (in fact, you *must* not do so).

Just be advised that you are mission out on a lot of fun if you don't.

This documentation is an extract of the corresponding chapters from the DML documentation and may contain references to “DML stuff” that initially may be difficult to understand. Just ignore it, and you should be fine. It's what people do with most of what I write anyway.

Check out the demo missions “demo – Bug Hunt” and “demo – debug events and more” which are configured to have the debugger running and see what it can do for you.

Have fun creating fantastic missions!

-ch

Zürich, January 1st, 2024

2 The Debugger

What is a “debugger”? That is geek-speak for a utility that is designed to help identify, and then hunt down, design flaws in your mission *while the mission is running*. The goal is that you then, once that you understand the flaw and piece together what is going wrong, you can eliminate that flaw in Mission Editor.



(8 years old) godson's impression of a debugger

DML comes with a fully-fledged **interactive** tool to help you track down bugs in your missions. Missions in DCS use surprisingly few (meaning they manage to accomplish impressive results with very little) methods to control a mission's flow. Even if you take into account (and DML's debugger *does*) the occasional script author, all flow of control in DCS is accomplished with only

- *Flags* - the doodahs that have a name and a number value
- *Events* - pre-defined situations like a player landing, or a unit being created
- *Tables* - posh “flags” that have more elaborate names and values

Everything in your mission comes down to those few things. In fact, DML *entirely* consists of tables and the (very common) ability to read and change flags, plus the ability to react to certain events.

Plain-vanilla missions use flags as their method to control the flow of a mission. In non-DML missions, designers place trigger zones, and then create trigger rules to perform actions when (and only when) certain conditions are met, which usually results in them to change a flag.

DML modules merely extend this and make much of it transparent; they make flags easier to understand through the concept of abilities that use ‘inputs?’ and ‘outputs!’ to talk to each other and depict flags as means to connect outputs to inputs.

One of the biggest problems that *all* mission designers face while they play-test their missions is that DCS does not provide them with easy means to inspect their flags, nor be notified when important stuff (from the viewpoint of the mission) happens. To make matters much worse, mission designers have no ability to intervene while a mission is running. They can't, for example, force an issue by changing a flag, nor spawning a unit – once the mission runs, all a designer can do is watch.

The Debugger changes all that, and more: it is heavily focused on providing you with comprehensive, easy-to-use, **interactive** (i.e., while a mission is running) and above all meaningful tools to inspect, track, analyze and change things. Many things.

Even better, The Debugger is designed with DML's way of mission authoring in mind, and it has integration right into the mission design phase: while you are wiring up DML modules you can use attributes to mark things that the debugger should watch for you – so that when the mission runs, the Debugger already knows what to look for.

In other word, there are two interlocking phases for debugging / play-testing a mission:

- **Set-up / Design Mission** – this is while you create your mission in Mission Editor. Whenever you feel like it, you can add attributes to trigger zones that tell the debugger about things to watch out for. You can set up your debugging session in your own time in Mission Editor, and have all information ready at your finger tips when you start the mission
- **Execution / Run Mission**– this is *while your mission is running*. Here The Debugger comes into its own, and hands you an array of powerful tools to hunt down even the most obscure of bugs. Ordinarily, you would only be able to observe. With The Debugger you can interactively intercede.

The Debugger's abilities fall into a couple of distinct features that I will list by increasing awesomeness:

- Observe and Report Flag Changes. This includes the ability to only bother you when a flag changes to a certain value.
- Interactively tell The Debugger to watch flags
- Interactively change the value of flags
- Report Events (e.g. report "dead" events)
- Interactively tell The Debugger which events to report
- Interactively remove (destroy) units
- Interactively spawn units, objects and effects where you have clicked on the F10 map
- Inspect any Lua Table inside the mission scripting environment
- Create and change Lua Tables inside the mission scripting environment

IMPORTANT NOTE:

To use the debugger interactively, you must enable the "F10 Map User Marks" mission option.

2.1 Using The Debugger (Summary)

The Debugger only “works” (is active) while a mission is running, but you can streamline your debugging session if you collect the information that you are interested in during your mission design phase in Mission Editor. A big part of mission debugging involves observing flags, tables and events. While each mission can use a myriad of either, for any given mission there are only a few that are really of interest to you.

I designed The Debugger while using it to debug (of course) DML-based missions, and it became quickly obvious to me that it could be used *much* easier if there was a way for me to prepare it before I test the mission that I’m working on, to set The Debugger up to focus on the information that I’m interested in. Since one of DML’s biggest advantages is its ability to read mission-specific information from attributes in trigger zones, it was a natural way for me to set up The Debugger for the next test session. Doing so can save you a lot of time and repetitious work but is by no means required.

2.1.1 Adding The Debugger to your mission (DML)

You add The Debugger to your mission just like any other DML module: by adding it as a DOSCRIPT or DOSCRIPTFILE action after “dcsCommon” and “cfxZones” AT START.

NOTE

There is a (less powerful), non-DML version of The Debugger that I made for our less fortunate friends who do not use DML. That version is called “THE DEBUGGER STANDALONE”. Never mix that standalone version into your DML-enhanced mission, or unpredictable things can happen.

2.1.2 Adding The Debugger STANDALONE to your mission (non-DML)

Add “theDebugger standalone 2.1.lua” as a DOSCRIPT or DOSCRIPTFILE to your mission AT START.

2.1.3 During design (Mission Editor)

This step is optional, and only serves to save you, the mission designer, time and nerves because – let’s face it – there will be more than one test session, and test sessions are the only known disproof of Einstein’s definition of insanity being “doing the same thing over and over and expecting different results”.

Name	Value
events?	1-6, 30-33, 15



```
*** monitoring events defined in <Events to monitor>:
monitoring event <$_EVENT_SHOT = 1>
monitoring event <$_EVENT_HIT = 2>
monitoring event <$_EVENT_TAKEOFF = 3>
monitoring event <$_EVENT_LAND = 4>
monitoring event <$_EVENT_CRASH = 5>
monitoring event <$_EVENT_EJECTION = 6>
monitoring event <$_EVENT_UNIT_LOST = 30>
monitoring event <$_EVENT_LANDING_AFTER_EJECTION = 31>
monitoring event <$_EVENT_PARATROOPER_LANDING = 32>
monitoring event <$_EVENT_DISCARD_CHAIR_AFTER_EJECTION = 33>
monitoring event <$_EVENT_BIRTH = 15>
```

In Mission Editor you can, simply by adding attributes to trigger zones tell The Debugger

- Which flags to observe and note when their values change
- Which events to observe and to tell you about should they occur
- Which units to spawn in-game should you give the appropriate command.


Of course you can tell The Debugger which events or flags to observe while the mission is running. It's a lot more work, though.

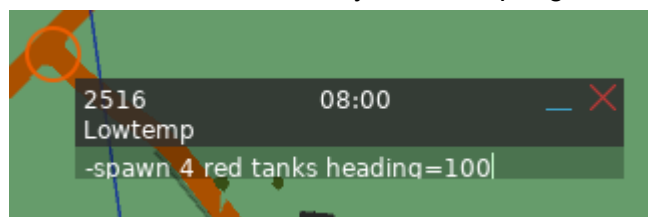
So, when you are gearing up to test your mission, you can add those flags that are most important to a "debug?" attribute (anywhere), and do the same for the "events?" that you are interested in. When the mission starts up, The Debugger transfers that data into its workspace and starts working for you. How? Read on.

2.1.4 In-Mission

The Debugger is **interactive**, meaning that at any time while the mission is running, you can interact with it: tell it to track flags, inspect tables, spawn or remove units etc. DCS was not designed for that kind of interactivity (it lacks what people call a 'console' – an interface to enter commands and see their results), and therefore The Debugger uses a slightly clumsy, if creative approach: it utilizes the in-game "Mark Label" mission function as console. You enter all commands to the debugger through that function, and all results are displayed on-screen through the standard "Text to all" mission output.

So, when you are debugging a mission, The Debugger is actively watching what you may enter for Mark Label text, and if what you type matches one its expected keywords, it responds accordingly. A typical mission debugging session runs as follows:

- The mission starts up, The Debugger loads, and sets up accord to any attributes that you may have added to trigger zone with Mission Editor.
- It watches  events and flags as instructed and tells you when something happens that you have told it to watch out for
- The Debugger also watches whenever a player enters something into the "Mark Label" game function, and if it matches one of its command keywords, it springs into action, executing the command that the player has invoked. These commands can range from adding or removing flags at and from watch lists, so spawning units, or even changing "tables" in the mission scripting environment – ad hoc changes to the mission



This combination of "watching for certain things to happen" and being able to intercede at any time by issuing game-changing commands is what makes The Debugger so powerful.

You would normally use The Debugger to

- Verify that things happen in the order that you have envisioned while you designed the mission. For this you monitor flags and tables to ensure that they change when, and in the way, that you have anticipated.

- Artificially force situations that you may have not foreseen. For this you change flags, spawn units or remove units.
- Check what happens (the sequence of things that happen) when a certain situation arises – again by monitoring flags and tables.
- Make situations arise (very useful to test otherwise difficult to test for edge cases). For this you change flag values, set table values, spawn units or remove units at will.
- Investigate and document how DCS handles certain situations to make sure that your assumptions are valid, and that your mission handles this correctly.
- If you have enabled persistence, you can also export the debugging log to a text file for better analysis later.

2.1.5 Debugger commands (overview)

Using the “Mark Label” F10 Map tool, players can issue various commands to The Debugger. All commands start with a hyphen “-” and are executed when the *player clicks outside* the mark text box (i.e. *not* when pressing “enter”)

2.1.5.1 Changing and Observing Flags

Command	Description
show	<p>Syntax</p> <pre>show <flagname/observername></pre> <p>Display the current value of all flags that are observed by <observername>. If there is no observer <observername>, the value of the flag named <flagname> is displayed.</p> <p>Examples:</p> <pre>show samStatus</pre> <p>Displays the current values of all flags that are observed by the observer “samStatus”. If there is no such observer currently defined, the value of the flag named “samStatus” is displayed.</p>
set	<p>Syntax</p> <pre>set <flagname> <number></pre> <p>Set the flag named <flagname> to the numeric value of <number>.</p> <p>Examples</p> <pre>set samsAlive 1</pre> <p>This sets the value of the flag named “samsAlive” to 1 (the number one)</p>
inc	<p>Syntax</p> <pre>inc <flagname></pre>

Command	Description
	<p>Add 1 (one) to the flag named <flagname>'s current value.</p> <p>Examples</p> <pre>inc samsAlive</pre> <p>This adds 1 to the current value of the flag named samsAlive. If the current value is 5, then the new value will be 6</p>
flip	<p>Syntax</p> <pre>flip <flagname></pre> <p>Set the value of the flag named <flagname> to either 0 or 1. If the current value is 0, then the new value is 1. If the current value is anything but 0, the new value is 0.</p> <p>Examples</p> <pre>flip samsAlive</pre>
observe o	<p>Syntax</p> <pre>observe <flagname> [with <observerName>]</pre> <p>Add the flag named <flagname> to the observer <observername>. If you omit <observername>, the flag is added to The Debugger's default internal observer.</p> <p>Flags that are under observation are checked periodically for a change in value. Should the value change, The Debugger reports the change.</p> <p>Examples</p> <pre>observe samsAlive o samsAlive</pre> <p>Adds the flag samsAlive to the internal observation list</p>
forget	<p>Syntax</p> <pre>forget <flagname> [with <observerName>]</pre> <p>Remove the flag named <flagname> from the observer <observername>. If you omit <observername>, the flag is removed from The Debugger's default internal observer.</p> <p>Examples</p> <pre>forget samsAlive</pre> <p>Removes the flag samsAlive from the internal observation list</p>
new	<p>Syntax</p> <pre>new <observername> [[for] <condition>]</pre>

Command	Description
	<p>Create a new observer named <observername> that you can add flags to observe to. If you omit the “with <condition>” part, the observer is created to watch for “change” in the flag’s value. The Debugger understands all current DML Watchflag conditions</p> <p>Examples new watchingU with >3 new watchingU >3</p> <p>Creates a new observer named “watchingU” that is set to trigger if a flag’s value changes, and the new value is greater than 3 (three)</p>
update	<p>Syntax</p> <pre>update <observername> [to] <condition></pre> <p>Change the condition that trigger flags watched by the observer named <observername> to <condition>.</p> <p>Examples update watchingU to change update watchingU change</p> <p>Changes the trigger condition for the observer named “watchingU” to “change”, i.e. any change in value</p>
drop	<p>Syntax</p> <pre>drop <observername></pre> <p>Remove the observer named <observername></p> <p>Examples remove watchingU</p>
list	<p>Syntax</p> <pre>list <match></pre> <p>List all observers whose name contains <match></p> <p>Examples list ching</p> <p>Lists all observers that have “ching” in their name, e.g. “watchingU”</p>
who	<p>Syntax</p> <pre>who <flagname></pre>

Command	Description
	<p>List all observers are tracking (observing) the flag named <flagname></p> <p>Examples</p> <pre>who samsAlive</pre> <p>Lists all observers that (perhaps among other flags) observe the flag named "samsAlive"</p>
reset	<p>Syntax</p> <pre>reset [<observername>]</pre> <p>Re-load all values of the flags that are currently observed by observer <observername>. If you omit <observername>, all observers are reset. This command is useful prior to re-starting The Debugger when you have suspended it during testing, as it reloads all flags at their current values, avoiding false positives when The Debugger resumes.</p> <p>Examples</p> <pre>reset watchingU</pre> <p>Lists all observers that have "ching" in their name, e.g. "watchingU"</p>

2.1.5.2 Analysis

Command	Description
snap	<p>Syntax</p> <pre>snap <observername></pre> <p>Create a "snapshot" of all flag values that are observed by observer <observername>. This snapshot can later be used to compare to current values to see which flags have changed. If you omit <observername>, a snapshot of all currently observed flags is created.</p> <p>Examples:</p> <pre>snap watchingU</pre> <p>Creates a snapshot of all flags that the observer "watchingU" is observing.</p>
compare	<p>Syntax</p> <pre>compare</pre> <p>Compare all flags from the last snapshot with their current values and provide a table with past and current values; flags that have changed in value are marked</p>

Command	Description
	<p>Examples:</p> <pre>compare</pre>
note	<p>Syntax</p> <pre>Note <any remark></pre> <p>Send <any remark> to The Debugger's output, allowing you to annotate it (useful only if you plan to export the output for later analysis)</p> <p>Examples:</p> <pre>note <-- before landing of A-10</pre> <p>Inserts "<-- before landing of A-10" into The Debugger's log</p>
save	<p>Syntax</p> <pre>save <file name></pre> <p>Save the contents of The Debugger's log to a plain text file with the name "<filename>.txt"</p> <p>Examples:</p> <pre>save Sinai Tourist Test</pre> <p>Requires that</p> <ul style="list-style-type: none"> • The persistence module is loaded and • Your DCS installation is de-sanitized <p>If writing fails or the persistence module is missing, The Debugger will complain, and then continue working as if nothing happened.</p>

2.1.5.3 Spawning and Removing Units, Objects or Effects

When you spawn units, objects, or effects with TheDebugger, they appear in-game at the center of the Mark Label, i.e. where you clicked on the F10 in-game Map

Command	Description
spawn	<p>Syntax</p> <pre>Spawn [<number>] [<coalition>] <type> [heading=<number>]</pre> <p>Spawn <number> instances of <type> belonging to <coalition> into the game. The spawned units have a heading of number. If you omit</p> <ul style="list-style-type: none"> • <number>: 1 (one) unit is spawned • <coalition>: the units belong to NEUTRAL • heading=<number>: the units head 000 (North)

Command	Description
	<p>To get a list of all allowed types and their current settings, use the special command</p> <pre>spawn ?</pre> <p>Examples:</p> <pre>Spawn 4 red tank</pre> <p>Creates 4 tanks belonging to RED at the position of the Mark Label, heading North (000). The units that are created for the “tank” type are defined by the tape name that you can pass to The Debugger with a trigger zone; by default, it is a “T-90”. To see which type is currently set to what, use the ‘spawn ?’ command</p>
remove	<p>Syntax</p> <pre>remove <name></pre> <p>Remove the group/unit/object named <name> from the game. If you don’t know the name of an object, unit, or group, you can use the F-10 map and click on it.</p> <p>Examples:</p> <pre>remove Lander-1-1</pre>
smoke	<p>Syntax</p> <pre>smoke <color></pre> <p>Add smoke with the color <color> at the location of the Mark Label. The colored smoke times out after 5 minutes</p> <p>Examples:</p> <pre>smoke red</pre>
boom	<p>Syntax</p> <pre>boom <number></pre> <p>creates an explosion of strength <number> on ground level at the point of the Mark Label. If you omit <number>, an explosion of strength 1. Note that a strength of 3000 is usually sufficient to level a block of buildings in a town.</p> <p>Examples:</p> <pre>boom 20</pre>

2.1.5.4 Events

The Debugger provides you with comprehensive “event” monitoring. “Events” are pre-defined (by ED) things that happen during a DCS mission, and that a mission can be notified of. Each “Event” is identified by a number, for example an Event of ID = 4 means that an aircraft has landed. The Debugger can tap into these events and give you notice whenever events happen. It also can recall the last event that it was instructed to look for, and perform a more detailed analysis on the event itself.

Note that events are an advanced mission design topic.

Command	Description
eventmon	<p>Syntax</p> <pre>eventmon [all off <number> last ?]</pre> <p>Use the event monitoring ability. If you omit all options, The Debugger defaults to monitoring all events.</p> <p>Examples:</p> <pre>eventmon off – remove all events that are monitored eventmon all – add all events to monitor list eventmon 4 – add event ID 3 (landing) to monitor list eventmon last – perform analysis on the last recorded event eventmon ? – show all events that are currently monitored</pre>

2.1.5.5 Inspecting and Changing Tables

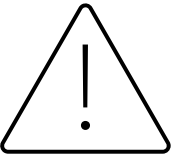
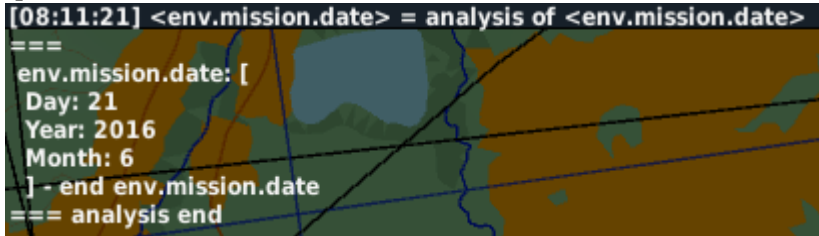

While flags and events make out some 99% of most mission designer’s mission debugging work, some designers also need to get into the nitty gritty of design, and require access to the deeper abyss of the mission scripting environment – they want to inspect (and possibly change) tables.

Note:

the following is not for the faint of heart. You have been warned.

The Debugger provides you with commands to inspect, analyze and, yes, change Tables. This is the no-holds-barred section of The Debugger, where you can show your meanest streak; The Debugger won’t blink and stay at your side – at least until you sink the entire mission environment.

Command	Description
q	<p>Syntax</p> <pre>q <fully qualified table name></pre> <p>Return the value of a Mission Scripting Environment table.</p> <p>Examples:</p> <pre>q env.mission.theatre</pre>

Command	Description
	<p>Returns the value of the env.mission.theatre table. With standard missions, this returns the name of the map (a string) that the mission is run on, e.g. "Caucasus".</p> <p>If the type of the queried table is a number or string, the value is returned immediately, otherwise the type is returned. You can then use the 'a' (analyze) command to get a detailed description of the table's content.</p>
<p>a</p> 	<p>Syntax</p> <pre>a <fully qualified table name></pre> <p>Analyze a Mission Scripting Environment table. Recursively analyse the entire table and visualize the structure as text. Some tables are really big, and The Debugger is not easily frightened, so beware telling it to analyze, for example, "env". Because it will.</p> <p>Examples:</p> <pre>q env.mission.date</pre>  <p>Analyses the structure and lists all node values of the env.mission.date table.</p>
<p>w</p> 	<p>TREAD CAREFULLY – THIS IS NOT A DRILL</p> <p>Syntax</p> <pre>w <fully qualified table name> [=] <any Lua expression></pre> <p>Assigns (and executes if required) <any Lua expression> to <fully qualified table name>. <any Lua expression> can be anything legal in Lua, including full programs.</p> <p>Examples:</p> <pre>w dp = {x=1, y=2}</pre> <p>Creates a table with fields x = 1 and Y = 2 and assigns it to the new global table dp.</p>

2.1.5.6 Miscellaneous

Command	Description
start	<p>Syntax</p> <p><code>start</code></p> <p>Restart a stopped debugger. If The Debugger is already running, this has no effect. Before you start a stopped debugger, consider the reset command to re-load all currently tracked flags</p>
stop	<p>Syntax</p> <p><code>stop</code></p> <p>Stops the active parts of The Debugger, meaning this command simply stops tracking events and flags. You still can issue commands as before.</p>

2.2 Interactive Use during Missions

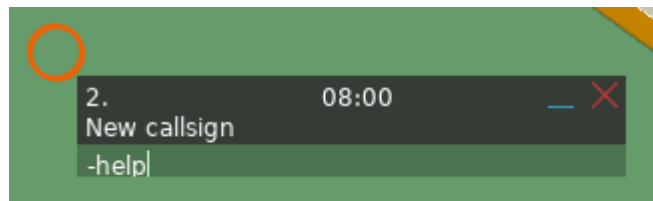
You can issue commands to the debugger through an interactive text entry field (it's normally used for different things, and we re-purpose it in DML)

In a mission that uses the debugger, you go to the Map View (F10), and click on the "Mark Label" button on the tool bar, then click somewhere on the map.



"Mark Label"

A small orange circle appears, with a black title bar and a grey text entry field beneath. To enter a debugger command, click into the text field, and start typing. All debugger command start with a hyphen ('-' a.k.a. minus-sign). In the example on the right, I have entered the '-help' debug command that causes the debugger to give you a list of all interactive commands it knows.



To activate the command, you must click outside the text box.

The debugger instantly responds with a text message:

- an error message when it doesn't fully understand the command. In that case, the Map Mark remains open, and you can correct the command by clicking into it.
- If command is accepted, the Map Mark disappears, and the debugger responds with a text message.

PRO TIP

DCS has a handy 'MESSAGES HISTORY' function that you can access by pressing 'ESC' during the mission. All messages, including all debugger messages are accessible from here and you can use this to read past messages that have already faded from the screen.

2.2.1 Remembering it for you wholesale: -help and -?

Should you ever forget which commands do what, simply entering '-help' or '-?' prompts the debugger to display a short helpful list of all commands and their uses. It's not much but it goes a long way.

```
-show <flagname/observername> -- show current values for flag or observer
-set <flagname> <number> -- set flag to value <number>
-inc <flagname> -- increase flag by 1, changing it
-flip <flagname> -- when flag's value is 0, set it to 1, else to 0

-observe <flagname> [with <observername>] -- observe a flag for change
-o <flagname> [with <observername>] -- observe a flag for change
-forget <flagname> [with <observername>] -- stop observing a flag
-new <observername> [[for] <condition>] -- create observer for flags
-update <observername> [to] <condition> -- change observer's condition
-drop <observername> -- remove observer from debugger
-list [<match>] -- list observers [name contains <match>]
-who <flagname> -- all who observe <flagname>
-reset [<observername>] -- reset all or only the named observer

-snap [<observername>] -- create new snapshot of flags
-compare -- compare snapshot flag values with current
-note <your note> -- add <your note> to the text log

-spawn [<number>] [<coalition>] <type> [heading=<number>] | [?] -- spawn
units/aircraft/objects (? for help)
-remove <group/unit/object name> -- remove named item from mission
-smoke <color> -- place colored smoke on the ground
-boom <number> -- place explosion of strenght <number> on the ground

-eventmon [all | off | <number> | ?] -- show events for all | none | event <number> | list
-eventmon last -- analyse last reported event

-q <Lua Var> -- Query value of Lua variable <Lua Var>
-a <Lua Var> -- Analyse structure of Lua variable <Lua Var>
-w <Lua Var> [=] <Lua Value> -- Write <Lua Value> to variable <Lua Var>

-start -- starts debugger
-stop -- stop debugger

-save [<filename>] -- saves debugger log to storage

-? or -help -- this text
```

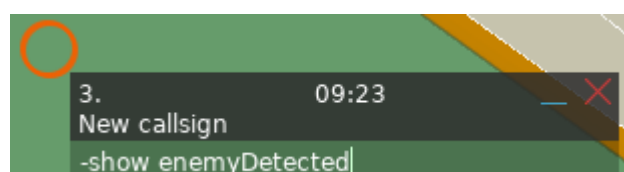
We'll go through all the commands below, so there's no need to study above; just nod in appreciation.

2.2.2 Fundamental commands: -show, -set, -flip and -inc

Since they are used to store states and signal changes, flags abound in DCS missions, and even more so in DML-based mission that use flags to communicate between modules. So, it is important to be able to look at flags, and change their value.

The Debugger has three easily understood commands for that:

- **-show <flagname>**
Accesses the flag <flagname> (e.g., 'enemyDetected') and displays its current value as a text message.
Note that all debugger messages

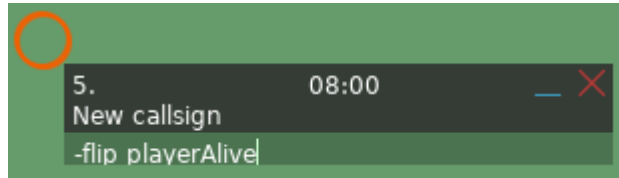


are time-stamped so you can (using Message History) reconstruct when (in mission time) the flag had that value.:

```
[09:24:57] flag <enemyDetected> : value <0>
```

- **-inc <flagname> and -flip <flagname>**

Most of DML's module inputs react to a *change* in a flag's value, meaning that mostly, they aren't looking for a particular value, merely a *change* in value. To conveniently support this, the debugger sports two quality of life features: commands to increment (inc) and 'flip' (when the flag's value is equal to 0, set it to 1, else set it to 0) a flag.

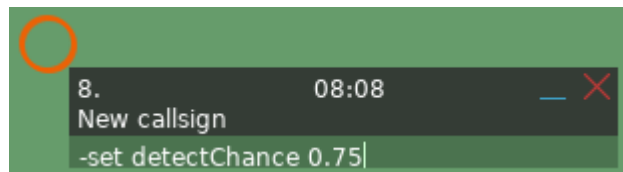


```
*** [08:00:28] debug: flipped flag <playerAlive> from <0> to <1>
```

This is usually sufficient to trigger any input for DML modules that have a Watchflag looking for 'change'.

- **-set <flagname> <value>**

With this you set the flag <flagname> to <value>. Note that values must be numbers. You can enter negative numbers and fractions (e.g., -3.141 is valid)



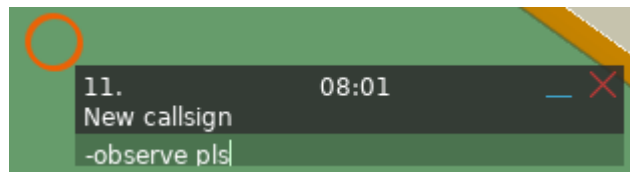
```
*** [08:10:07] debug: set flag <detectChance> to <0.75>
```

Be advised that DCS currently does not handle fractions for flag values well, and can/will convert fractions into integers without notice

2.2.3 Let the Debugger work for you: -observe and -o

On many occasions, it's sufficient to inspect the value of a flag to determine if everything is as you intend them to be. But some issues are more difficult to track down, and instead of constantly checking if a flag has changed its value, it would make more sense to be alerted when a flag that you are interested in changes its value.

And that's what "-observe" (or "-o" for lazy people) does for you: it adds that flag to its internal watchlist. From now on, every time a flag on that watchlist changes, you'll be notified immediately:



Note that there is no limit to the number of flags we can add to the debugger's watchlist, and we'll later look at 'observers' a nice quality of life feature that makes it easy to prepare and manage watchlists both interactively and from inside ME.

```
*** [08:00:21] debugger: now observing <pls> for change with <+DML Debugger+>.
```

For now, we just added 'pls' to the debugger's watchlist, and it patiently examines all flags and waits for them to change. Note that the debugger gives you more information than you probably realized:

- The time when observation started (08:00:21 in mission time)
- Flag name (pls) that it is now observing
- This flag is observed for 'change'
- A mysterious "+DML Debugger+" is watching the flag. We'll get back to this when we talk about observers

So the mission runs, and the flag 'pls' suddenly changes. This is what you may see

```
---debug: 08:04:08 -- Flag pls changed from 1 to 0 [+DML Debugger+]
```

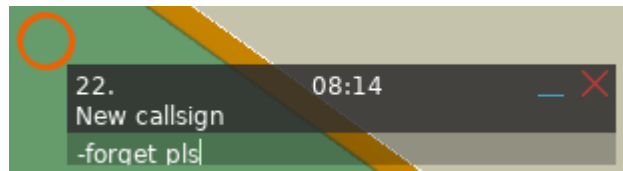
The debugger tells you:

- Mission time (08:04:08) when the change was detected. This can be helpful if you later need to establish the order in which the changes happened
- Flag 'pls' changed. This is important to know since usually you have the debugger watch multiple flags at the same time
- From 1 – the value that the debugger remembers from the last time it checked
- To 0 – the number it has now
- +DML Debugger+ - the 'observer' that noted the change. We'll come back to that later.

So, whenever you want to see if and when a flag switches values, have the debugger - observe it for you, and you'll find out as soon as it happens.

2.2.4 Losing interest: -forget

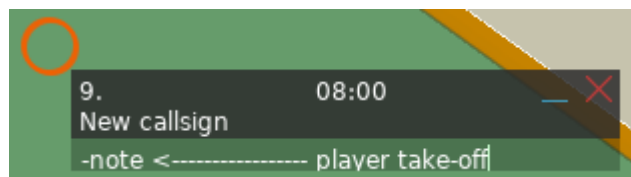
Sometimes, after you have seen your flag of interest change the way you intended, you may no longer need to be notified of its changes, want to focus on other flags, or simply de-clutter the text display on your right side. It's time to tell the debugger to -forget that flag. When successful, the debugger responds with



```
*** [08:14:48] debugger: no longer observing pls with <+DML Debugger+>.
```

2.2.5 Annotating Message History: -note

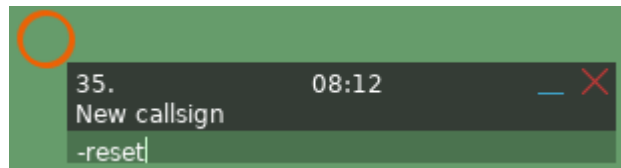
A lot of debugging has to do with looking at the message log (Message History) and finding the relevant entries. Thus, it may be helpful to find points of interest that you marked: a flag value that you found strange, or an event that you want to mark in the log and later find. For this, the debugger has a '-note' command. Anything you type after the command will appear as a text message in the log, making it easy to find later:



```
*** [08:00:24]: <----- player take-off
```

2.2.6 Clearing the Slate: -reset

When you are observing flags, you may occasionally want to bring all flags up to their current state without causing them to trigger a change.



You most commonly do that if you stopped the debugger during a mission, and then want it to continue without creating false positives for the flags it watches (when the debugger is stopped, it no longer tracks flags but continues to listen to interactive commands). To do so, simply use the -reset command, and the debugger loads all current values as the start values for any observed flag.

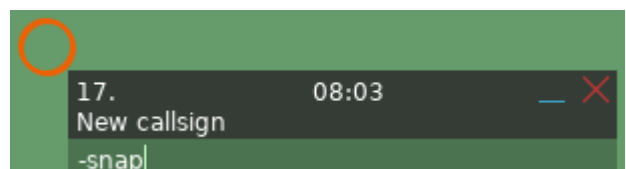
```
*** [08:12:37] debug: reset complete.
```

2.2.7 Getting the BIG picture: -snap and -compare

Up to now, we have talked about *individual* flags, and how the debugger can help you to track down changes. Often, though, when you design a mission, you have many flags that taken together reflect the state of your mission. Often, mission designers determine that something important should happen, when the state of a mission reaches a certain point, and that point is often expressed with a combination of flags.

For example, you may want to trigger the enemy's retreat if it has lost more than half of its aircraft, their two forward SAM sites are destroyed, and the player still has three aircraft left. All this can (and usually is) expressed via flags. If, during testing you notice that something goes awry, and you are losing track of the complex interdependence of flags, it's time to bring out the big guns: **snapshots**!

A snapshot is nothing more than looking at, and then noting down the values of all the flags that you are observing. This little thing is incredibly useful, because you can then later compare current values against those noted before and see which have changed. Using '-snap' you can take snap shots - at any time (as a matter of fact, the debugger automatically takes a snapshot at the very beginning of the mission); just remember that the debugger only keeps the most recent snap shot.

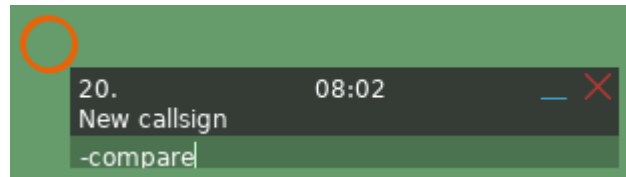


```
*** [08:05:12] debug: new snapshot created, 8 flags.
```

Note that the debugger's response tells you how many flags it is tracking in the snapshot. Make sure that you check this against your expectation, as there are few more frustrating situations when you find out that that one flag you needed to compare wasn't part of the snapshot.

So far, this wasn't impressive. *Now* comes the cool stuff:

At any time, you can tell the debugger to **'-compare' the snapshot to the current situation**. The debugger provides you with a nice table of past (snap) and current (now) values and marks those flags where the value has changed:



```
*** [08:14:47] debug: comparing snapshot with current flag values
<t2> snap = <0>, now = <0>
<t3> snap = <0>, now = <0>
<pls> snap = <0>, now = <0>
! <t1> snap = <0>, now = <6> !
! <5> snap = <0>, now = <2> !
<4> snap = <0>, now = <0>
<7> snap = <0>, now = <0>
<6> snap = <0>, now = <0>
*** END
```

As you can see, the values for flags “t1” and “5” have changed since the snapshot was taken, and they are marked with an exclamation point (“!”) before and after their data: “t1” was set to 0 in the snapshot and now holds the value of 6, while the (classic, numbered) flag “5” was recorded with a value of 0, and currently is set to 2.

2.2.8 Bringing in the (big) Guns: -spawn

The Debugger can spawn a lot of varied troops in a very short time. To do so, you issue the ‘-spawn’ command, and add some (optional) parameters that control how many units to spawn, what kind of troops to spawn, who they belong to, and which direction they are facing.

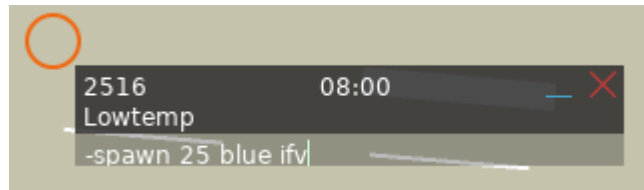
2.2.8.1 The spawn command

Spawning troops in The Debugger is very easy: simply give the command, and with the command you can give it some more info. In general, the command is structured like this:

```
-spawn <how many> <faction> <type> <heading=xxxx>
```

Now, except for the type parameter, all others are optional, and you can put them in any order. I simply like above order. So let's go through them one by one:

- *how many* [default = 1]
A number that tells The Debugger how many units of <type> it should spawn. If you spawn ground or naval types, there is no strict upper limit, and theDebugger will happily spawn 1000 units for you. If DCS can then handle the strain is something else entirely.



The units spawn in a grid formation, to pack as many units as possible in a close formation.

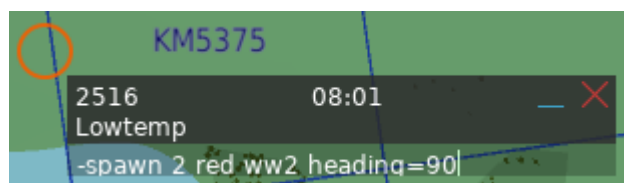
If the unit type is an aircraft, the number of spawned units is capped to 4 by DCS.

- *faction* [default: NEUTRAL]
The faction (red/blue) to whom the spawned units belong.
- *heading=0* [default:0]
No spaces, the word 'heading' and the equal sign are mandatory. The heading is heading in degrees, with 0 being North.
If present, all units will head in that direction. If omitted, all units will head North. If the spawned units are aircraft, they receive also receive a waypoint 100km in that direction.
- *type*
This is spawn's greatest ability. There are literally hundreds of different units in DCS, and it would be silly to assume that mission designers know their correct (arcane) type designation by heart. So The Debugger uses a system of **Generic Stand-ins**: you say 'tank', and The Debugger says 'T-90' (with "T-90" being DCS's correct internal unit typeString for a T-90 main battle tank). So, all you need to remember is the Generic name (see 'Spawned Generics, below), and The Debugger looks up the correct DCS unit typeName for you. Even better, you can tell The Debugger which in-game types to use for which generic name.

2.2.8.2 Spawned Generics

The Debugger knows the following generic names:

- *tank* – default T-90
- *ifv* – default BTR-80
- *inf* [infantry] – default Soldier M4
- *sam* – default Roland ADS
- *aaa* – default Zoo Shilka

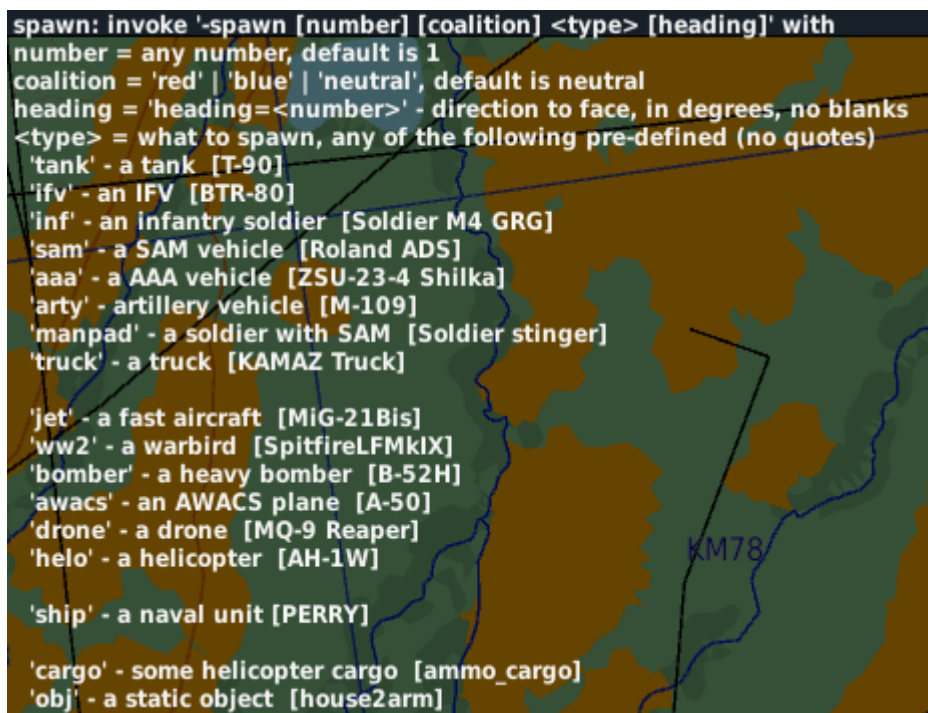


- *arty* [artillery] – default M-109
- *manpad* – default Stinger
Manpad
- *truck* – default KAMAZ Truck
- *jet* – default Mig-21
- *ww2* – default Spitfire
- *bomber* – default B-52
- *awacs* – default A-50
- *drone* – default Reaper
- *helo* – default AH-1
- *ship* – default Perry
- *cargo* [for helicopters] – default ammo
- *obj* [static object] – default Armed Watchtower



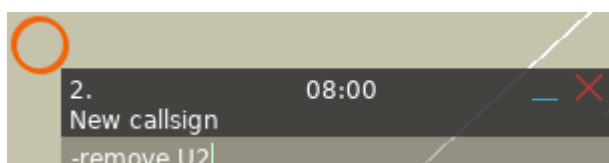
And if you can't remember which generics are available (and which in-game they are currently configured to spawn), you can always issue the *spawn help* command:

-spawn ?



2.2.9 Getting rid of 'em: -remove

You can use the debugger to remove units, groups or static objects by using the -remove command, and give the groups, unit's or static object's name.



Note that the name must be spelled correctly, including upper and lower case.

```
*** remove: removed unit <U2>
```

If the debugger can't find any Group, Unit or Object that matches the name you gave, it will return an error alerting you to that fact.

2.2.10 Smoke On! -smoke

Placing smoke to mark a location is easy with The Debugger. Simply issue the '-smoke' command, and optionally add a color (default is green). The smoke erupts for 5 minutes at the point of the Mark Label

2.2.11 Boom Baby! - boom

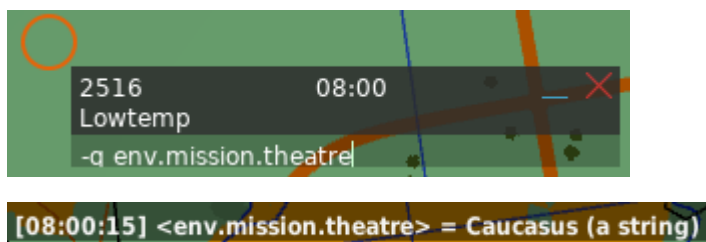
Similarly, you can place explosions with the -boom command. Except a smoke color, add an explosion strength (a number), and there's a big boom at the location of the Mark Label.

2.3 Killer Features

There are a couple of commands that take The Debugger somewhat outside the realm of a “mere mission debugger”: for advanced users, and especially mission script developers, there are commands to look at, analyse, and set Lua Tables. If you don’t know why this would be important, consider yourself lucky, and immediately proceed to the next section.

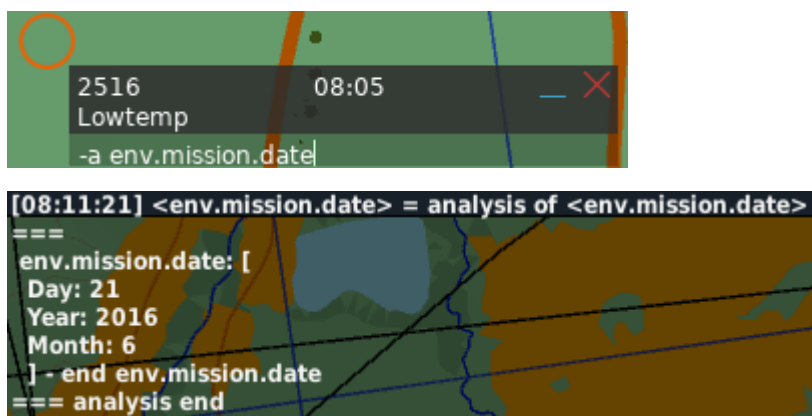
2.3.1 Q, no longer anon: -q

There are times when we need to inspect the value or gestalt of some Lua tables. The Debugger has the **-q** (for query) command that allows you to query *any* (fully) qualified Lua structure that exists inside the mission scripting environment. If the structure is simple, the value is shown immediately, else the type is shown.



2.3.2 Analyze this: -a

If the structure is a table, you can invoke the Debugger’s analyze **-a** ability, again with the (fully) qualified name. The Debugger’s analyzer recursively delves into the structure and visualizes the table and all node values:



2.3.3 W is for WARNING: -w

And now comes the feature that you were waiting for: Write, **-w** writes *any legal Lua expression* into any (fully) qualified table, creating a new table if and when required, overwriting existing ones. There are a couple of caveats:



2.3.3.1 Live Bullets Here

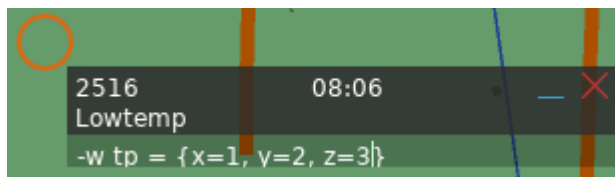
We aren’t in Kansas anymore, Dory. The Debugger shoots with live bullets, and like Mission Editor, there’s no Undo. If you change or overwrite a table, that’s that. If you accidentally crash the mission in the process, that’s often the *best case* scenario.

2.3.3.2 Yes, *any* legal Lua Expression

This is the other biggie. While you can use -w to create or change tables, that's only part of what The Debugger will happily allow you to do. Invoke methods while assigning a value? Have at it, Hoss! And yes, you can assign entire code stretches with that function. Why you should want to do this is anyone's guess, but it is possible.

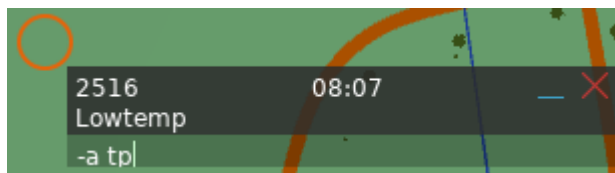
2.3.3.3 A Mild-Mannered Example

So, let's use this ability to first create a (global) table that we name 'tp', and add the attributes x, y, z to it, with different values each. Then we'll admire our work with the -a command:



```
[08:06:46] <tp> set to <{x=1, y=2, z=3}>
```

And then



```
[08:07:37] <tp> = analysis of <tp>
===
tp: [
y: 2
x: 1
z: 3
] - end tp
=== analysis end
```

So, now you have enough rope to hang yourself – enjoy the ride!

2.4 Big Brother: Observers

Most missions can be broken into separate, logical parts, and are designed accordingly. You may, for example, first design Blue's AAA, then ground defenses, then attack groups.

Usually, and especially with DML, they use flags to communicate. The debugger supports this and allows you to group flags with 'observers' that even provide integration with Mission Editor to set up a debugging session before the mission runs.

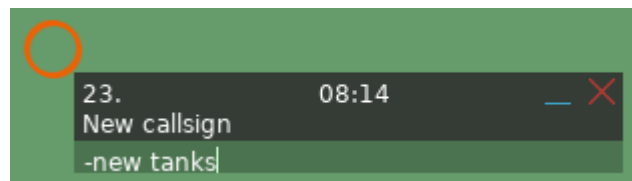
Besides being able to prepare a debug session in ME, observers offer another important ability that makes them tremendously useful in mission debugging: the ability to change what they report. By default, they notify you when a flag's value changes, just like we've seen above.

Observers, however, allow access to the full spectrum of DML's Watchflag ability, so you can easily tell an observer to only notify you if a flag has changed to a certain value. And **that ability makes using observers a killer feature**, especially in tandem with ME pre-setup: set up an observer that only notifies you if a certain set of flags attains a certain value – the closest you can get to “break points” in DCS without a source-level debugger.

2.4.1 Creating an observer: -new and new for

An “observer” is simply a List for flags that the debugger maintains for you. You can add flags to the list, and a flag can be part of multiple lists. By default, an observer behaves just like you have seen before: report when a flag on the list has changed its value.

You can give observers any name you like, **but to guarantee compatibility with all commands, make sure it does not contain any blanks**. I prefer short and meaningful names, because you will have to re-type the name every time you add a flag to it, and because it's always displayed with the flag name if the observer notifies you.



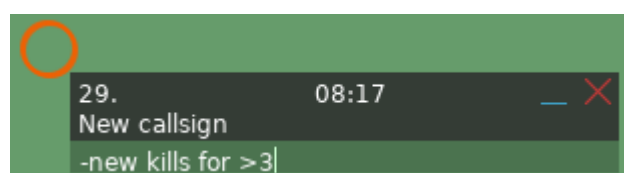
```
*** [08:17:26] debugger: new observer <tanks> for <change>
```

WARNING

Observer names that contain blanks can't have their condition assigned nor changed.

Note the 'for <change>' part in the debugger's response. This means that the new observer is set up to trigger (notify you) on “change”. This is the DML “change” Watchflag condition, and it is assigned by default to an observer. Since we did not add a condition when we created the observer with -new, the debugger automatically used “change” for the condition.

We can change that: Instead of simply issuing '-new <name>' we can add a DML condition (also called “Watchflag Method”) through the optional 'for <condition>' to the



observer. All flags that are added to this observer are checked against that condition.

The observer we create with “-new kills for >3” will check all flags that are added to this observer that, when their value changes, their new value is greater than 3, and if so, trigger a notification

```
*** [08:23:14] debugger: new observer <kills> for <>3>
```

NOTE

Remember that flags can be added to multiple observers, so you can **have the debugger check a flag** for you **against multiple conditions simultaneously**, a great relief when debugging complex missions!

2.4.2 Adding flags to observers: -observe with

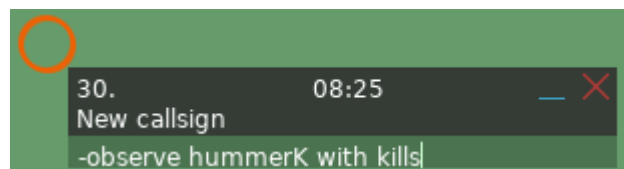
You add flags to an observer with the ‘-observe’ command.

Wait – didn’t we cover ‘observe’ before? Yes – and for a very simple reason: The debugger maintains a default observer, called “+DML Debugger+” that is set up to trigger on change. When you simply -observe a flag without also giving an observer, the debugger simply adds it to its own default observer. We already saw that when we first experimented with observing flags:

```
---debug: 08:04:08 -- Flag pls changed from 1 to 0 [+DML Debugger+]
```

Note the part in the brackets “[+DML Debugger+]” – whenever an observer notifies you of a flag that triggered its condition, the debugger will also tell you which observer spoke up. Here, it was the observer called +DML Debugger+, which is the one that the debugger happens to create when it starts up, and that all flags get assigned to when you don’t give another observer

So how do we add a flag to a specific observer? By adding ‘with’ to the command and add the name of the observer. Let’s add the flag ‘hummerK’ to the recently created observer ‘kills’ that triggers when the value of that flag changes to a value greater than 3.

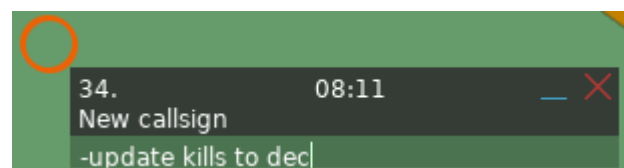


```
*** [08:02:28] debugger: now observing <hummerK> for value >3 with <kills>.
```

2.4.3 Changing an observer’s condition: -update to

Occasionally, you may want to change what an observer is looking for in the flags it is managing.

You can easily do this by using the -update command with the observer’s name and the new DML Watchflag condition (‘Method’). Let’s change the ‘kills’ observer so that it notifies you whenever the last value was larger than the new one – the ‘dec’ condition



*** [08:12:33] debugger: updated observer kills to <dec>

REMINDER:

Remember that an observer name that contains blanks cannot have their condition changed. So if you repeatedly get an error whilst trying to change a condition, verify that the observer name is free of blanks.

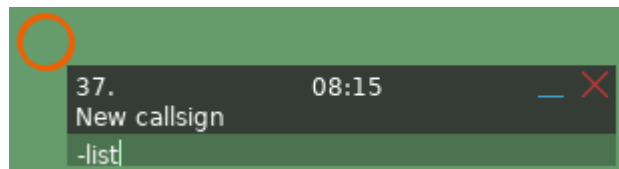
2.4.4 Supported Observer Conditions

Since the debugger is entirely based on DML, it inherits *all* DML abilities. The observers internally use DML Watchflags to trigger a notification, and hence we can use all “DML Watchflag Methods” in the debugger, just as if we are setting up trigger zones in ME:

- 'change' or '#'
- 'off' or '0' or 'no' or 'false'
- 'on' or '1' or 'yes' or 'true'
- 'inc'
- 'dec'
- 'lohi'
- 'hilo'
- '>(number)' or '>(name)'
- '=(number)' or '=(name)'
- '<(number)' or '<(name)'
- '#(number)' or '#(name)'

2.4.5 Show me what you got: -list

When your mission becomes more complex, and especially when you use debugger's ME integration, where you come into a mission with numerous observers already set up, it's helpful when you can get an overview of the observers that debugger currently knows, and what they are looking for. A list of all observers and their trigger methods is only a -list command way.



```
*** [08:00:25] listing all observers:
<Look for pls = 15> for <value =15> (1 flags)
<Look for t1= 4> for <value =4> (1 flags)
<rndObserver> for <value change> (5 flags)
<many flags> for <value change> (8 flags)
<+DML Debugger+> for <value change> (0 flags)
```

The list above shows a total of 5 active observers, the number of flags they each track, and what they are looking for in the flags they track. For example, the observer called “Look for pls = 15” triggers when a flag it observes changes its value to 15, and it is currently tracking 1 flag.

Here we also see our good friend “+DML Debugger+”, the debugger’s own default observer, which currently does not track any flags, and is set up to look for a change in values.

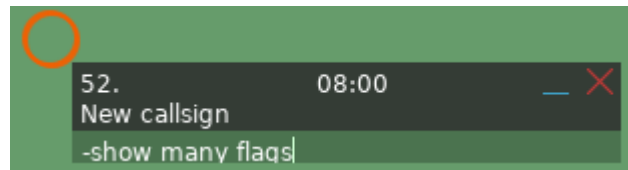
Also note that from the early mission time (25 seconds into the mission) it is obvious that we are looking at pre-set observers created in ME – more on that later.

Finally, note that many observers have a blank in their name yet use complex conditions - another sure indication that they were set up with ME

2.4.6 Show and tell: -show observername

Just like you can tell the debugger to show you the value of a single flag, you can tell it to show you all the flag values that are managed by an observer. Simply use the observer’s name instead of a flag name

and you’ll get a list of all flags that it is currently watching, along with their current values, and the condition that the observer looks for to alert you to a change:



```
*** [08:01:08] flags observed by <many flags> looking for <value change>:
! f:<t1> = <6> [current, state = <0>, HIT!]
f:<t2> = <0> [current, state = <0>]
f:<t3> = <0> [current, state = <0>]
f:<4> = <0> [current, state = <0>]
! f:<5> = <2> [current, state = <0>, HIT!]
f:<6> = <0> [current, state = <0>]
f:<7> = <0> [current, state = <0>]
f:<pls> = <0> [current, state = <0>]
```

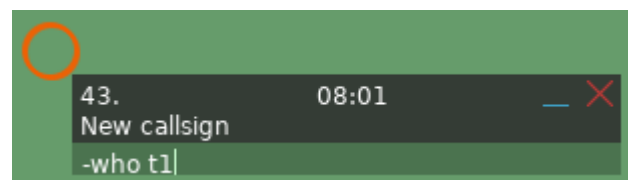
Note that the exclamation point in front and the text in square brackets (e.g. [current, state = <0>, HIT!]) are only relevant (and can only occur) when you have disabled the debugger. When current and state values diverge (indicated by the exclamation point and ‘HIT!’ legend), the debugger will produce a notification for that flag when it becomes active. This is so that you can the show command even when the debugger itself is inactive. Use the ‘-reset’ command before starting the debugger to avoid unnecessary notifications.

2.4.7 Who’s zoomin’ who: -who

When your mission becomes complex, and you have multiple observers tracking your flags, it may become helfult to find out which observers track a particular flag.

This is when you can employ the -who

command, which runs through all observers to tell you which observer is checking out that flag, and what it is looking for:

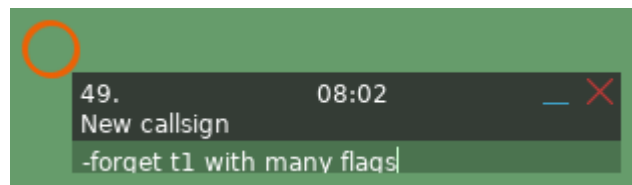



```
*** [08:00:38] flag <t1> is currently observed by
<Look for t1= 4> looking for <value =4>
<many flags> looking for <value change>
```

Above response shows that the flag “t1” is observed by two observers: “Look for t1=4” which has the (somewhat predictable) condition “=4”, and the “many flags” observer, which is looking for a change in value.

2.4.8 ... and Forget Me Nots: -forget with

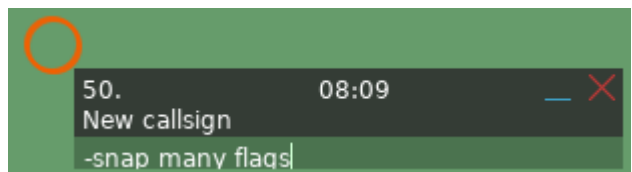
You can selectively ‘forget’ a flag from an observer by adding ‘with’ and the observer’s name to the ‘-forget’ command. This removes the flag from that observer list, and retains it on all other observers that also observe it.



```
*** [08:01:26] debugger: no longer observing t1 with <many flags>.
```

2.4.9 Oh, snap! -snap observername

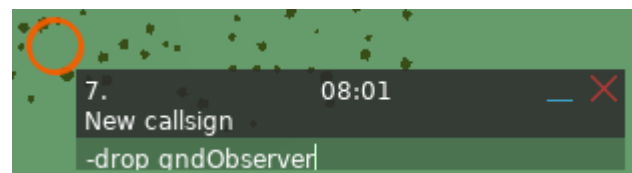
You can limit a snapshot to only the flags that are on one observer’s list. Simply add the observer’s name to the -snap command, and only the flags observed by that observer are recorded. So when you -compare them next time, only that smaller slice of flags are compared to their actual values. When combined with ME-based observer setup, you gain a comfortable, fine-grained and exact analysis tool for your mission.



```
*** [08:09:51] debug: new snapshot created, 7 flags.
```

2.4.10 The spotless mind: -drop observername

Observed flag changes troubling you? No problem. With -drop you can tell the debugger to instantly forget an observer. The only observer you cannot drop is the instant debugger’s own observer, and it will make you that you don’t try to drop it.



```
*** [08:02:25] debugger: dropped observer <gndObserver>
```

2.4.11 Main switch: -start and -stop

Finally, there are two commands that you can use to turn the debugger on and off at will: -start starts the debugger if it was inactive, and -stop will make it inactive (go on stand-by). When inactive, the debugger itself (the part that tracks flags) is dormant, but the daemon (the part that listens to your commands) is still very much active and responds to all commands. The debuggers observers can still be shown, edited etc.

2.5 Saving the Debugging Log: -save [filename]

The debugger has the ability to write the entire log of the current debugging session to disk so you can then use text tools to analyze the output at a later date, send it to other people (e.g. the mission designer) and compare it to other logs.

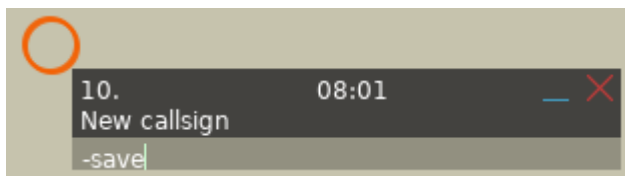
2.5.1 Using -save [filename]

When enabled (see below) you can, at any time, save the log from the current debugging session to your storage as a text file. The contents may look something like this (probably much longer), and reflect all debugging messages:

```
cfx debugger v1.1.0 started.
interactive debugDemon v1.1.0 started
  enter -? in a map mark for help
---debug: 08:00:10 -- Flag t1 changed from 0 to 2 [many flags]
---debug: 08:00:11 -- Flag 5 changed from 0 to 2 [many flags]
t1 is now equal to four! [08:00:12]
---debug: 08:00:12 -- Flag t1 changed from 2 to 4 [many flags]
---debug: 08:00:15 -- Flag t1 changed from 4 to 6 [many flags]
*** [08:00:19] debug: inc flag <seq> from <0> to <1>
---debug: 08:00:19 -- Flag seq changed from 0 to 1 [gndObserver]
---debug: 08:00:22 -- Flag gnd2 changed from 0 to 1 [gndObserver]
---debug: 08:00:27 -- Flag gnd1 changed from 0 to 1 [gndObserver]
---debug: 08:00:32 -- Flag gnd3 changed from 0 to 1 [gndObserver]
---debug: 08:00:38 -- Flag hog changed from 0 to 1 [gndObserver]
```

Note that the log only contains debug messages, it does not contain the entire message log that is available from the “Messages History” menu.

To save the log, simply issue a “-save” command. If you do not provide a file name with the save command, the file will be saved as “DML Debugger Log.txt” in your current “Missions/” folder (the one DCS uses in saved games).



```
+++debug: log saved to <C:\Users\ [redacted] \Saved Games\DCS\Missions\DML Debugger Log.txt>
```

Note that since the debugger uses DML’s ‘persistence’ module, you can change the save location (important for dedicated servers). Please refer to DML’s ‘persistence’ module documentation for details.

If a previous log of that same name exists at that location, it will be overwritten, replacing the contents with the new log.

You can provide your own file name for the log, e.g. “-save my debug log”, which will save the log to a text file named “my debug log.txt”. Note that if you provide any file extension other than “.txt”, a “.txt” will be added, so you might as well omit it altogether.

2.5.2 How to enable -save

In order to use '-save', you must add the 'persistence' module to your mission before the mission loads the debugger, and perform the necessary steps outlined in the 'persistence' section to allow a DCS mission to write data to your storage.

2.5.3 Do I have to add persistence?

If you do not add 'persistence' to your mission, '-save' is disabled and you receive a reminder of that fact every time you try to use '-save'. There is no need to add persistence unless you really want to use that feature.

2.6 Integration With Mission Editor (Optional)

Crucially, the debugger provides full integration with ME, so you can set up observers at your leisure in ME, and have them instantly at your fingertips when your mission is-mission running.

Without ME integration, you'd spend the first minutes of each bug hunt setting up observers, add flags, and possibly miss crucial flag events that happen at the very beginning of the mission. In ME, on the other hand, you can take all the time in the world to set up observers, edit them to your heart's content, and most importantly, only have to set them up once, for as many mission runs later as you want.



Of course, using this feature is optional and entirely up to you. As soon as you add the debugger to your mission, it's good to go: press 'Fly Mission' and you're off. If you merely intend to hunt down a single, particularly obstinate bug, that may be a viable approach: start debugging right away before you bring in the ME-based debug artillery.

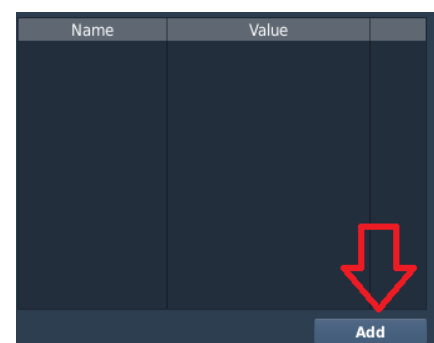
Note that using debugger ME integration is trivial when you already have worked with DML - and may feel strangely comfortable otherwise.

2.6.1 Creating Observers in ME

If you have any experience with DML, the next step is not only familiar, it also takes only a few seconds: it uses standard trigger zones and an attribute name that you'll never forget. If you have never worked with DML, these steps may feel strange at first, but they will become natural soon.

To create an observer in ME, simply

- Create a Trigger Zone anywhere on the map. We use the trigger zone's 'attributes' that we can edit directly in ME to pass information to the debugger, so the location of the zone is irrelevant. Place it where it's convenient to access and not in the way of other mission details. Likewise, the zone's size and color aren't relevant, choose any.
- Add a new attribute to the zone by clicking on the Add button in ME's Trigger Zone editor (the one that opens when you create, or click on, a trigger zone)
- Rename this attribute 'debug?' (do not forget the question mark at the end). This marks the zone as an observer, and when the debugger starts up when you run the mission, it reads all the data you put into the 'value' field.
- Then add the flag names that you want that observer to observe as values to the attribute. You can list as many flags as you like, and the individual flags must be





Name	Value	
debug?	seq, gnd1, gnd2, gnd3, goHc	

separated by comas. If you are using classic, numbers-only flag name, you can also add entire ranges (e.g., 10-17) of flags as part of the list. In the example on the left, we have added the flags 'seq', 'gnd1', 'gnd2', 'gnd3' and 'goHog' as flags to this observer.

The observer inherits the zone's name as its own name in the debugger, and you'll immediately see it with -list when you start the mission.

You can also set the observer's trigger condition with the "debugTriggerMethod" (or one of its synonyms, like "sayWhen") attribute. It defaults to 'change' but any

Name	Value	
debug?	pls	
debugTriggerMethod	=15	

DML Watchflag Method is valid. Please be advised that terms like "DML Watchflags" and "synonym" are DML parlance that will be inaccessible to you until you try your hand at DML. You can use the debugger without knowing that stuff, you merely can't max out its usefulness like DML warriors can.





WARNING:

Remember that the interactive debugger disallows you to change the condition for an observer with a name that contains blanks. If you intend to be able to change an observer's trigger condition while the mission is running, do not use blanks in the trigger zone's name.

2.6.2 Dedicated and Stacked Zones, tracking local flags

While it's a good idea to create dedicated (stand-alone) 'debug zones', i.e., Trigger Zones that only use the "debug?" attributes, you can easily add the "debug?" to other zones and "stack" the debug module onto other DML modules that use the zone.

This may be not as clean as a dedicated debug zone, but it is an easy fix if you want to quickly debug a particular zone's flag, especially if that zone uses local flags that are difficult to access otherwise.

Name	Value	
radioMenu	Pulser	
itemA	Start Pulser	
A!	*startPulse	
itemB	Stop Pulser	
B!	stopPulse	
debug?	*startPulse	



In the example to the right, we have stacked the 'debug?' attribute with a "radioMenu" module. the debugger now automatically tracks the local "*startPulse" flag with this zone's observer, and a notification appears whenever the player chooses the "Start Pule" radio item (read the RadioMenu module description to find out how it works).

2.6.3 Activating Event Monitoring

Very similar to how you tell The Debugger from Mission Editor which flags to observe, you can use attributes to tell it to monitor events for you: simply use an attribute named "events?" and list all the events that you are interested in. From the moment that the mission starts, The Debugger will monitor them and tell you which of them occur. This is especially helpful

in situations where you expect events to occur at, or very near the start of the mission and you may miss them otherwise.

Also, if you add a “verbose = true” attribute to the same trigger zone, The Debugger reports all the events that it is picking up to monitor from that zone (both as a reminder and to verify that the correct events are monitored at mission start):

Name	Value	
events?	1-6, 30-33, 15	
verbose	yes	

*** monitoring events defined in <Events to monitor>:
monitoring event <S_EVENT_SHOT = 1>
monitoring event <S_EVENT_HIT = 2>
monitoring event <S_EVENT_TAKEOFF = 3>
monitoring event <S_EVENT_LAND = 4>
monitoring event <S_EVENT_CRASH = 5>
monitoring event <S_EVENT_EJECTION = 6>
monitoring event <S_EVENT_UNIT_LOST = 30>
monitoring event <S_EVENT_LANDING_AFTER_EJECTION = 31>
monitoring event <S_EVENT_PARATROOPER_LANDING = 32>
monitoring event <S_EVENT_DISCARD_CHAIR_AFTER_EJECTION = 33>
monitoring event <S_EVENT_BIRTH = 15>

2.6.4 Setting up Generic Stand-ins

You can again use a trigger zone to set up which units The Debugger spawns.

Create a trigger zone, and re-name it

“debuggerSpawnTypes” (spelling and capitalization is significant).

Then add attributes and name them after the generic type, and set the value to the DCS type name (see here

<https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB>) that you want The Debugger to spawn instead. You only need to add attributes for those generics that you want to replace.

In the example above, we replace the type that The Debugger usually spawns for the ‘inf’ generic (a “Soldier M4”) with the better-looking (and animated) “Soldier M4 GRG”. In that mission, all invocations of “-spawn inf” now result in spawns of the “Soldier M4 GRG”.

Name	Value	
inf	Soldier M4 GRG	



2.7 Adding The Debugger to your mission

The debugger can be added to your mission in two different ways:

- Stand-alone (for non-DML-enhanced Mission)
- As standard module for a DML-enhanced Mission


The stand-alone version is provided for those who have never ventured into the fun world of DML enhanced mission scripting, and simply need a debugger NOW!!!!one please. If you are already using DML, please use the DML-version. The stand-alone version may lack features of the DML version and is not maintained as regularly.

2.7.1 Stand-Alone (NO DML in your mission)

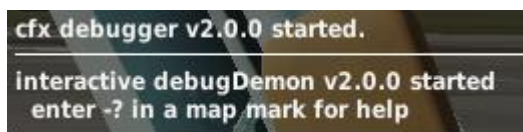
I don't need to tell you that you are missing out on a lot of fun, so let's get this thing rolling for you as quickly as possible. To add The Debugger (standalone) to your mission

- Open the mission in ME
- Add a "trigger rule" to your mission:



- Click on the set rules for trigger icon 
 - Under the ① Triggers heading, click on the NEW button
 - From the new fields that show up, click on Type: 1 ONCE pop-up menu and change that to 4 MISSION START
- TYPE: 4 MISSION START
- On the far right under the ② ACTION Heading, click NEW
 - On the Action pop-up, click and select DO SCRIPT FILE
- ACTION: DO SCRIPT FILE
- Click on the OPEN button and select "Mission Debugger.lua"
 - Save the mission.

From now on the debugger runs whenever the mission starts. You'll now that the debugger is active when you see a message similar to the following when the mission starts:



Note that the stand-alone debugger supports adding observers from ME just like the DML version.

2.7.2 DML-enhanced Missions

If you are already using DML in your mission, adding the debugger is straightforward. Just add The Debugger.lua like you would any other module and make sure that you are also using 'dcsCommon' and 'cfxZones'. It would be strange if your mission didn't already include them, so it will take you roughly 10 seconds to add the debugger. Make sure you are using the newest versions of dcsCommon and cfxZones.

WARNING

NEVER ADD A STAND-ALONE DEBUGGER INTO A DML-ENHANCED MISSION!

The stand-alone version may lag a few versions behind, and usually comes packaged with older versions of dcsCommon and cfxZones that *will* conflict with your DML-modules.

2.7.3 ME Attributes

2.7.3.1 Debug Zones (for setting up Observers)

Name	Description
debug?	List the flag names that the debugger is to observe. All flags listed here are accessible from the debugger under the observer with the same name as the trigger zone MANDATORY
triggerMethod debugTriggerMethod inputMethod sayWhen	Trigger condition for the flags (the observer's "condition" that triggers a report for the flag) Defaults to 'change'
method outputMethod debugMethod	DML Method for the debugger's output flags. Rarely used. Defaults to "inc"
notify!	DML flag to bang! when a flag listed in debug? triggers
debugMsg	Message to output when a flag listed in debug? triggers. Supports wildcards, including <f> for the flag name that triggered, and <z> for the zone name. Note that this allows you to provide individual message formatting per observer , a feature that is not available for the interactive debugger. Defaults to "---debug: <t> -- Flag <f> changed from <p> to <c> [<z>]" which results in a message similar to <code>---debug: 08:00:12 -- Flag t1 changed from 2 to 4 [many flags]</code>

2.7.3.2 Event Tracking

Name	Description
events?	Comma-separated list of the events IDs (numbers, e.g. "4" for landing) that The Debugger should track when the mission starts. Supports ranges (e.g. "3-9") Example 1, 3-5, 12 MANDATORY
verbose	If set to true, The Debugger reports which events are added to the watch list:

Name	Description
	<div>monitoring event <S_EVENT_LAND = 4></div> <p>Defaults to false (no reporting of monitored events)</p>

2.7.3.3 Generics

To tell The Debugger which unit type to spawn for a generic

- Place a Trigger Zone on the map in ME
- Name it “debuggerSpawnTypes” (note: name must match exactly)
- Add any of the following attributes to this zone (all are optional) and enter the type string for a unit as described here: <https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB> :

Name	Type Name
tank	default T-90
ifv	default BTR-80
inf	Infantry, default Soldier M4
sam	default Roland ADS
aaa	default ZSU-23-4 Shilka
arty	[artillery] default M-109
manpad	default Soldier Stinger
truck	default KAMAZ Truck
jet	default Mig-21Bis
ww2	default SpitfireLFMkIX
bomber –	default B-52H
awacs	default A-50
drone	default MQ-9 Reaper
helo	default AH-1W
ship	default Perry
cargo	[for helicopters] default ammo_cargo
obj	[static object] – default house2arm

verbose	<p>If set to true, all changes to generics are logged when the mission starts:</p> <pre>+++debug: changed generic 'Inf' from <Soldier M4> to <Soldier M4 GRG></pre> <p>Defaults to false, no logging of generic changes</p>
---------	---

All entries are optional

2.7.3.4 Debugger Configuration

To configure the debugger module via a configuration zone,

- Place a Trigger Zone somewhere in ME
- Name it “debuggerConfig” (note: name must match exactly)
- Add any of the following attributes to this zone:

Name	Description
verbose	Show even more debugging information. Default is false
active	If the debugger is active on start-up. Defaults to true (debugger is active at mission start)
on?	DML Watchflag to activate/start the debugger if it's inactive/stopped Defaults to <none>
off?	DML Watchflag to deactivate/stop the debugger if it's active/running Defaults to <none>
reset?	DML Watchflag that when triggered causes the debugger to re-baseline all flag values with current values. Useful when you are about to start the debugger after it was deactivated to avoid false positives. Defaults to <none>
state?	DML Watchflag that when triggered causes the debugger to log all currently observed flags and their values Defaults to <none>
ups	Number of updates per second, the “time resolution” of the debugger. Defaults to 4 (every 0.25 seconds)

2.8 Bug Hunt - A Live demo

So let's run the debugger through its paces. Fire up DCS ME and load the mission "demo – Bug Hunt".

We use this demo mission to explore many features of the debugger. Some of the examples may be a bit contrived, but they should give you a good indication how you can get the best out of the debugger: both in ME and in your mission. Oh yeah... it's also show-boating some of DML's abilities, but what the heck. I wanted to have some fun, too

This demo documentation takes a slightly different approach from most of DML's demo-docs: we are looking at individual use cases instead of the entire demo.

2.8.1 Please ignore me: Self Test

Run the mission, enter the Frog's cockpit and do nothing. After a few seconds, a few lines of text should appear on the right side, reporting some flag changes. This is a small self-test added to this mission to check the debugger's integrity. This is part of the **mission**, set up using ME triggers; the self test is **not** part of the debugger. If you want to test the debugger in your missions, you'll have to add your own tests. The object of this self test is to verify that the debugger version that is added to this mission is working correctly. After it ran successful you can safely ignore it. I'm merely using it as a quick quality check.

2.8.2 Observing and setting flags to debug your mission

Let's imagine a Herc transport flight 'Elvis' from Batumi to Kobuleti that the player is tasked to protect. We can activate the Herc with a flag, which prompts it to spawn on Batumi runway 31 and takes off. When the transport has left the airport zone, we spawn an aggressor. Should the Herc survive and land at Kobuleti, the mission is won.



To control the mission, we are using the following flags:

- *goElvis*
A flag that activates the Herc on the runway. It'll then take off after a brief interval
- *elvisLeaving*
This flag turns true when the transport 'Elvis' leaves the 'Home Plate' zone around Batumi (All of group outside). Elvis is airborne at that point and turns towards

Kobuleti. When this flag turns true, we want to spawn a red Albatross (the albatross is harmless so we can test the remainder of the logic, it will be switched out later for a more dangerous flight)

- *elvisSafe*
This flag turns true when the transport arrives at Kobuleti. It signals a successful conclusion of this mission. Remaining attackers should turn home, and some congratulatory message should be displayed. We simply trigger a message in the demo
- *goAlba*
A flag that activates the aggressor Albatross. This flag turns true when Elvis has been activated with goElvis and elvisLeaving is true

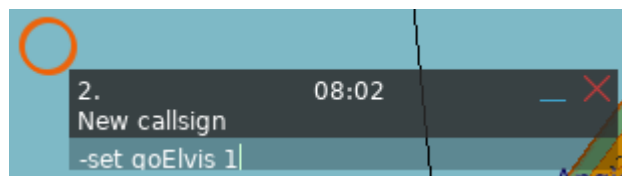
So, let's run the mission. Initially all seems well: We have a cool Radio Menu wired into flag 'goElvis' (provided by a DML radio menu zone), but we will be using the debugger just because we can.

Start the mission and hop into the Hog or Frog. Look outside. A couple of vehicles around a hog (we'll get to that cool animation later). The runway is clear, the sky is clear. All correct so far

A couple of messages start crawling on the screen, from some pre-set observers that we ignore for the moment (we'll get to those later)

Now let's activate the Herc. We could use the radio menu Other→Elvis→Start Elvis (which would increment goElvis from 0 to 1), but we want to use the debugger to brute-force this.

We go to F10, place a mark (put it near Batumi runway 31 so you can see what is happening on the map as well) and enter "-set goElvis 1". Immediately we are rewarded by the Herc appearing on the runway.



So all is good. Except – no. When we step through all aircraft, we notice that the Albatross, too, has spawned. Why?

"-show goAlba" reveals that indeed, goAlba has been triggered:

```
[08:00:20] flag <goAlba> : value <1>
```

Ok, how did that happen?

Restart the mission. Now, **before we spawn elvis**, we "-observe goAlba"

```
*** [08:00:15] debugger: now observing <goAlba> for value change with <+DML Debugger+>.
```

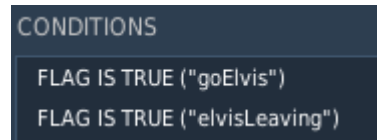
Now start Elvis with "-set goElvis 1"

And... As soon as we set goElvis to 1, the debugger reports a change to goAlba: it goes from 0 to 1, triggering the Albatros to spawn.

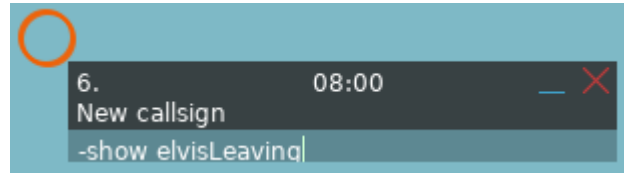
```
*** [08:00:47] debug: set flag <goElvis> to <1>
--debug: 08:00:48 -- Flag goAlba changed from 0 to 1 [+DML Debugger+]
```

Why???? --

goAlbatros is supposed to turn to true only when Elvis exists (goElvis was given, which causes it to spawn), and elvisLeaving is true (which can only turn true when the unit is outside of the home plate zone).



Let's have a look. Restart the mission, and "observe goAlba" as before. But since we are now suspicious, let's also examine "elvisLeaving" before we spawn Elvis. We expect this value to be zero, since Elvis hasn't left yet.



[08:01:31] flag <elvisLeaving> : value <1>

Ouch. It's 1, not 0 as we had expected!

In hindsight that's correct. elvisLeaving is defined as



Since the Elvis group has not yet spawned, it's also entirely outside of the home plate zone, and the flag immediately goes to 1. So here's your exercise: How would you fix that?

2.8.3 Setting flags to skip/advance mission stages

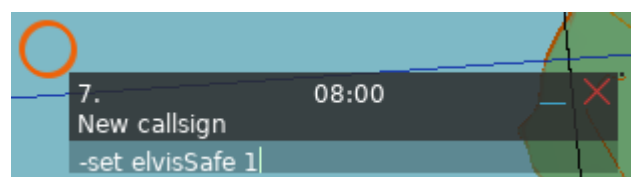
The next use case is so obvious that I only mention it for completeness:

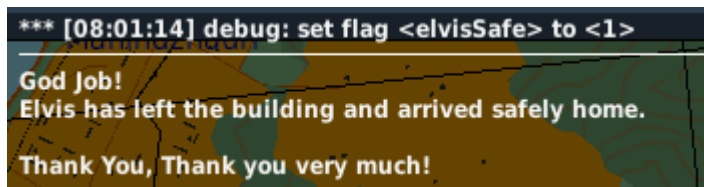
A common mission design pattern is to divide a complex mission into smaller, more manageable 'phases' like start-up, assemble, ingress, attack, egress, and recovery. For the mission to know which phase it is in (i.e., which action it needs to trigger), flags are used. Using a debugger to trigger these phases whenever you want can save you a lot of time, because you no longer must successfully navigate all previous phases. Simply set up the flags to the values you know they need to have, and then trigger that phase [pro tip: also try to intentionally set up flags so they *don't* match the phase's expectation, trigger it and see if it can deal with this gracefully – a technique called negative-testing)]

In our example, the flag goElvis starts the phase where Herc takes off, and when goAlba becomes true, the enemy attack phase is triggered. A final phase is triggered when the Herc arrives at Kobuleti: the mission is successful.

So let's debug this phase without having to wait for the Herc to arrive:

The final phase starts when the Herc arrives at Kobuleti and this causes the elvisSafe flag to turn to 1. Since we have a debugger all we need to do is brute-force the flag. We can do this immediately at mission start – we don't even need to start the Herc for this.





So, the final phase works correctly – except for thy typo in “Good” all is working well.

2.8.4 Using Observers and snapshots to debug your mission

It's usually the small things that make the difference between a good and a great mission. You can have a good mission where the AI takes off and attacks ground forces. And you can sweat the details, and add stuff that doesn't change the mission, but how it's experienced. Here's a small detail that adds flavor to a mission.

We have a Hog sitting on the ground, tasked and ready to go. But now, just to add some flavor, it's also surrounded by some support vehicles. When it's time to go, the vehicles all withdraw, and the plane starts taxiing to the runway.



To add this to your mission, you'd likely set up a group each for the ground vehicles. Each group has a move order, and that order is suspended with a 'Hold' order until a flag is set. When it's time to go, you set the flags that allow the vehicles to move, and when the AI aircraft is no longer blocked, it starts moving. The specifics here aren't important [except that it's trivial with DML, and a headache without], and we see that we have a number of flags that all must change before the plane can move.

In our demo, the three groups Ground-1 through Ground-3 are controlled with flags 'gnd1', 'gnd2' and 'gnd3'. We are also using a flag 'hog' to activate the hog [Note: we are using some *serious* DML magic in this example that reduces this task to little more than child's play. See the notes at the end of this section]. The entire sequence is started when flag 'seq' changes its value to 1.

So, we need to make sure that once flag 'seq' changes, flags gnd1, gnd2, gnd3 all fire before flag 'hog' fires. To do this, we set up our own observer to look at these flags and report their change. Let's assume that, somehow, behind the scenes our mission has some logic to delay changing these flags, one after the other, and perhaps even randomize them so the order isn't always the same (DML does this for us in this mission, but let's focus on the big picture).

So, let's create an observer and then start the sequence. Run the mission, and issue the following commands:

- -drop -gndObserver [NOTE: this is necessary because this observer already exists from a ME pre-set; we merely erase and then re-build it manually to hammer home the message just how much simpler ME pre-sets are]
- -new gndObserver [creates a new observer named 'gndObserver' that triggers on value change]
- -o seq with gndObserver [adds seq to gndObserver. -o is short for -observe]
- -o gnd1 with gndObserver
- -o gnd2 with gndObserver
- -o gnd3 with gndObserver
- -o hog with gndObserver

We now have an observer that looks at these flags for you. Let's make sure it's set up correctly with "-show gndObsever"

```
*** [08:21:06] flags observed by <gndObserver> looking for <value change>:
f:<seq> = <0> [current, state = <0>]
f:<gnd1> = <0> [current, state = <0>]
f:<gnd2> = <0> [current, state = <0>]
f:<gnd3> = <0> [current, state = <0>]
f:<hog> = <0> [current, state = <0>]
```

Now, we want to make sure that all these flags fire once we hit 'seq', and the easiest way to do that is with a snapshot, so we also take a snapshot of this observer:

"-snap gndObserver"

```
*** [08:21:13] debug: new snapshot created, 5 flags.
```

Five (5) flags are in the snapshot, which tracks with our expectation, so we are good to go.

We now start the sequence, change to F2 outside view, look at the vehicles and hold our breath:

"-set seq 1"

Text messages start appearing, and vehicles start moving away from the hog!

Then the Hog comes to live, and starts moving!

(you may also have noticed something strange: labels suddenly appear when units come to life – this is another DML feature that we ignore. Just marvel at the possibilities you have when you used DML)

```
*** [08:01:23] debug: set flag <seq> to <1>
---debug: 08:01:24 -- Flag seq changed from 0 to 1 [gndObserver]
---debug: 08:01:26 -- Flag gnd2 changed from 0 to 1 [gndObserver]
---debug: 08:01:30 -- Flag gnd1 changed from 0 to 1 [gndObserver]
---debug: 08:01:36 -- Flag gnd3 changed from 0 to 1 [gndObserver]
---debug: 08:01:42 -- Flag hog changed from 0 to 1 [gndObserver]
```


So we see that yes, the sequence appears to be somewhat random (gnd2-gnd1-gnd3), and they all happen before hog, which is good. Since the number of flags is rather small and easy to remember, we also know that all flags have fired that should have fired. But imagine you have a whole big list of flags that all should have changed by now.

That is when we use the snapshot compare command to get a nice overview, with helpful marks on those flags that have changed:

“-compare”

```
*** [08:02:26] debug: comparing snapshot with current flag values
! <gnd3> snap = <0>, now = <1> !
! <gnd2> snap = <0>, now = <1> !
! <hog> snap = <0>, now = <1> !
! <seq> snap = <0>, now = <1> !
! <gnd1> snap = <0>, now = <1> !
*** END
```

And yes, indeed. All flags in the snapshot have an exclamation point, as we expected. So, this startup sequence seems to work, the debugger has confirmed that for us.

2.8.5 Setting up Observers in ME

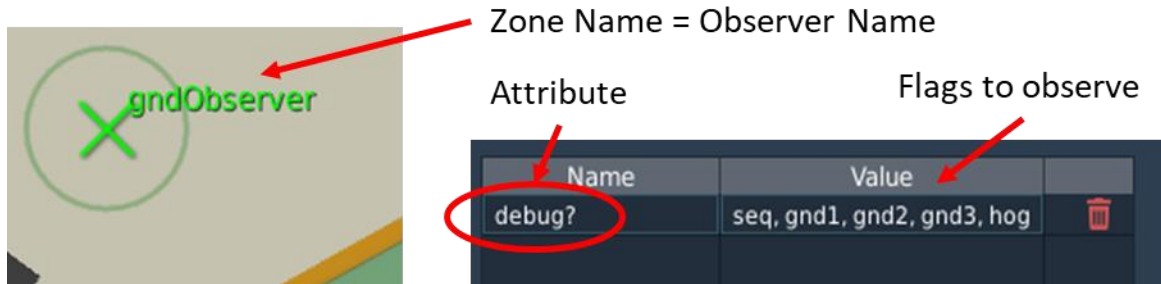
In the previous section, we manually built ‘gndObserver’, which we then used to track several flags and observed how they changed over time.

While building the observer, you also probably noticed that building even a small observer engendered some tedium, bordering on annoying. Moreover, it’s OK to do it once, but having to do this multiple times is a truly dread-inducing proposal. Which you will have to do, because when you restart the mission, all previous settings are gone, making you start all over. Perish that thought. In an unmarked grave.

That’s why The Debugger, having full access to DML’s zone manipulation capabilities, gives you the option to pre-set observers from within ME. Not only is setting up observers in ME lightning fast, it also persists – you save it with your mission and the observers are ready for you when the mission starts.

All you need to do to build an observer is to create a trigger zone anywhere on the map. In the trigger zone editor, change the name of that trigger zone to something you like – this will be the name of the observer, so be careful not to include blanks in the name if you want to be able to change the trigger condition during the live mission.

Once you have set the name, add a new attribute: set the attribute’s name to “debug?”. This will tell DML that it should pass the value of this attribute to the debugger. So, accordingly, set the value to a list of the flags that you want to observe: seq, gnd1, gnd2, gnd3, hog



Look at the (green) trigger zone 'gndObserver' in the mission. You see it's exactly set up like I described above: 'debug?' attribute with the flag names listed. If you run the mission now and "-show gndObserver", you'll get the exact same information as the observer we built with so much effort from within the live debugger.

2.8.6 Debugging local flags, flag concurrency

DML supports zone-local flags – flags that are “only visible inside the zone itself”. Of course, these flags aren't really inaccessible outside of the zone, they are merely difficult to access by accident, which is good enough for mission design purpose. As documented, a zone-local flag merely has a unique name composed out of the zone's name, an asterisk, and then the local name. The debugger knows how to hand zone-local variables and can track them.

The easiest way to track a zone-local variable is by adding the 'debug?' attribute to the zone that is home to the local variable. That also conveniently creates an observer for that zone, giving you full access to that flag when the mission starts.

Let's inspect the mission's 'sequencer' zone, the zone that directs the show of vehicles moving away from the Hog.

What we see are a pulser stacked on a flag randomizer. The pulser is started with 'seq' (very little surprise there) which feeds into 'startPulse?'. Once the pulser starts, it sends signals to the local '*go' flag on the pulse! Output.

That local flag feeds into the randomizer's rndPoll? input and triggers a new random cycle.

Note also the 'method' attribute, which is shared between the pulser and randomizer, setting both output methods to 'inc' (a pulser usually would default to 'flip').

Name	Value	
pulse!	*go	
pulses	4	
time	4-7	
startPulse?	seq	
RND!	gnd1, gnd2, gnd3	
rndPoll?	*go	
remove	yes	
rndDone!	hog	
onStart	no	
method	inc	

So, let's track the local *go" variable in the debugger. All we need to do is add a "debug?" attribute, with "*go" (with the asterisk) as value. Save the mission and run it. The debugger loads the new observer and immediately activates it.

Name	Value	
debug?	*go	
pulse!	*go	
pulses	4	
time	4.7	

Let's start the sequence with '-set seq 1' and see what happens

As soon as we '-set seq 1', gndObserver (which is also active since it's preloaded from the mission as well) response with a note that seq has changed.

We also see that flag *go (the local) flag has gone from 0 to 1. The only way to know which zone this flag belongs to is by looking at the end of the line, and we see the observer's name (which matches the zone name) in square brackets: [sequencer]. So, we know that sequencer's local go flag went from 0 to 1

```
*** [08:01:23] debug: set flag <seq> to <1>
---debug: 08:01:23 -- Flag seq changed from 0 to 1 [gndObserver]
---debug: 08:01:25 -- Flag *go changed from 0 to 1 [sequencer]
---debug: 08:01:26 -- Flag gnd3 changed from 0 to 1 [gndObserver]
---debug: 08:01:32 -- Flag gnd2 changed from 0 to 1 [gndObserver]
---debug: 08:01:32 -- Flag *go changed from 1 to 2 [sequencer]
---debug: 08:01:39 -- Flag gnd1 changed from 0 to 1 [gndObserver]
---debug: 08:01:39 -- Flag *go changed from 2 to 3 [sequencer]
---debug: 08:01:43 -- Flag hog changed from 0 to 1 [gndObserver]
---debug: 08:01:43 -- Flag *go changed from 3 to 4 [sequencer]
```

Looking down further we see that signals on *go always coincide with signals on the flags gndX and hog. Which brings us to another important observation:

Since we know that signals flow from the pulser to the randomizer, and the randomizer then sends out signals on the randomized flags, we know that the *go signal always happens before the randomizer's output signal.

The record, however, seems to indicate otherwise. In the amber box on the right, signal "gnd2" is listed before "*go", making it seem as if it happened before

```
---debug: 08:01:25 -- Flag *go changed from 0 to 1 [sequencer]
---debug: 08:01:26 -- Flag gnd3 changed from 0 to 1 [gndObserver]
---debug: 08:01:32 -- Flag gnd2 changed from 0 to 1 [gndObserver]
---debug: 08:01:32 -- Flag *go changed from 1 to 2 [sequencer]
```

*go. It did not – This is merely a result of how the debugger works, and the time "resolution" it has. By default, the debugger samples all flags four times per second, and everything that happens in that same time slice it reports as concurrent – it does not know what came first, and simply reports changes in the order it walks through the flags. Since gndObserver's list of flags is worked down before starting on sequencer's, it reports gnd2's change before it reports *go's.

Therefore, when you try to establish the order in which flags are set, always consider the observer's reporting time (red box). **If two signals have the same time stamp, you can't determine which signal came first** – to the debugger they happened concurrently.

Note also that simply increasing the debuggers ‘time resolution’ (for example by slimming the time slice to 0.1 seconds, there is no guarantee that this can give better results: many DML modules run on the same clock, and therefore there’s a good likelihood that the time interval between the pulser sending it’s message and the randomizer responding to the signal is less than 0.001 seconds. Keep the debugger’s resolution at 0.25 seconds (or more) to keep a good balance between accuracy and performance.

2.8.7 Now hear this: debugMsg and sayWhen observer attributes

When you set up observers in ME with the debug? attribute, you can change the message that the observer outputs. In addition to all special formatting that the messenger module supports, you can also use the following text wildcards:

- <c> for the flag’s current value (the value it has now)
- <p> for the flag’s previous value (the value it had last time the debugger sampled it)
- <f> for the flag’s name.

By default, the debugMsg attribute defaults to

```
“---debug: <t> -- Flag <f> changed from <p> to <c> [<z>]”
```

which returns a message similar to

```
---debug: 08:00:12 -- Flag t1 changed from 2 to 4 [many flags]
```

Look at the green “Look for t1=4” trigger zone



Name	Value	
debug?	t1	
sayWhen	=4	
debugMsg	<f> is now equal to four! [<	

It has two additional attributes. debugMsg is set to to a different format than normal:

```
“<f> is now equal to four! [<t>]”
```

That is what triggers the output

```
t1 is now equal to four! [08:00:12]
```

Which sticks out like a sore thumb – intentionally. You can use this feature to make an observer’s notifications pop out from the other notifications so you can quickly home in on them.

Note

that the only way to change an observer’s message is through ME, the debug demon has no such ability.

This zone also changes the observer's trigger condition from the default 'change' Watchflag condition to "=4" by using 'sayWhen'. This observer will therefore only notify you when the flag's value changes, *and* the new value is equal to 4 (see DML Watchflag definitions).




2.8.8 Mission Discussion: What DML does in this demo

While this mission's focus is on The Debugger, it's also a nice showcase for some of the modules that come with DML, and what they can do.





Of particular interest are three modules that we are using for the start-up sequence of the ground vehicles and AI-controlled hog:

Impostors

All the vehicles and the A-10 are turned into impostors (static objects) when the mission begins (onStart is true). This saves some CPU, and allows us to 'pause' the Hog's orders. The impostors are turned into AI-controlled units with their *reanimate?* Input, which the "gnd1", "gnd2", "gnd3" and "hog" flags are wired into.

Name	Value	
impostor?	gimp	
reanimate?	gnd1	
onStart	yes	

The A-10 also has a 'blink' attribute of 0.1 seconds that briefly removes it from the game before replacing the static object with the AI-controlled unit to avoid DCS's internal airfield slot controller assigning the A-10 a different starting position.

Name	Value	
impostor?	imp	
reanimate?	hog	
onStart	yes	
blink	0.1	

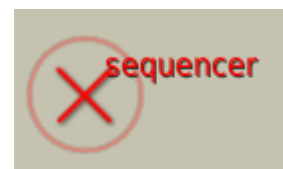
The impostors also will respond on their *impostor?* Inputs (flags "gimp" and "imp"), so if you were to issue '-set gimp 1' on the debugger after they have moved, you can turn them all back to static objects.

The Vehicle Startup Sequence

To make the mission look more natural, we want the three vehicles leave the Hog before the hog comes to life. To add some panache, we want to randomize the order in which the vehicles start moving, and also somewhat randomize the time between the vehicle's departure.

This is done with two modules: a Pulser that sends out several pulses with variable timing between them, and a Randomizer that randomly picks a flag from a list of flags and then sends a signal on that flag.






We also stack the two modules on the same zone 'sequencer', allowing us to use a local flag "*go" to communicate between Pulser and Randomizer without polluting this missions flag name space.



The Pulser

We want to start three vehicles, and the hog, making it a total of four (4) pulses that we want the pulser to produce for us. This is reflected by the 'pulses' attribute which switches it from the default 'infinite' to 4.

We also want a the time between individual activation to vary between 4 and 7 seconds, and that is what we find as value for the 'time' attribute (again replacing the default 1 fixed interval). We also want the pulser to wait until it's started (onStart is set to false), and we send a signal to start on the startPulse? Input via the now very familiar 'seq' flag.





Name	Value	
pulse!	*go	
pulses	4	
time	4-7	
onStart	no	
startPulse?	seq	

Finally, since we know that the only recipient of the Pulser's signals is the Randomizer that's also housed in the same zone, we use a zone-local flag "*go" to send output from the pulse! output.

The Randomizer

We store the flags for all the ground vehicles in the RND! array, allowing the randomizer to pick one each time it receives a signal on the rndPoll?" input, which is wired to the Pulser via local "*go". We use the Randomizer's ability to

randomly 'choose and discard' flags from its flags (remove = true), so that after sufficient poll requests have been received, it runs dry. We also use it's ability to signal that all flags have been 'used up' via the 'rndDone!' output. When this happens, we know that all vehicles have received their 'reanimate?' signal, and we can wire this signal directly into the Hog's Impostor reanimate? input, ensuring that it's always the Hog that starts last.

RND!	gnd1, gnd2, gnd3	
rndPoll?	*go	
remove	yes	
rndDone!	hog	

Shared flags and attributes

Since we are stacking modules in a zone, we can take advantage of shared attributes and flags, and we do so here. First, we use the zone-local "*go" flag to pass signals from the Pulser to the Randomizer.

Also, we are using the shared attribute name 'method' to set both the Pulser's

and the Randomizer's output method to "inc". Although, strictly speaking this isn't necessary (the default output methods for the Pulser ("flip") and Randomizer ("inc") would have worked), we include it in this demo just to show off.

method	inc	
--------	-----	---

2.9 Debug events and More (.miz)

This demo mission showcases some of the new features that I added to The Debugger in version 2, namely event monitoring, spawning and, of course, some Lua voodoo.

2.9.1 Starting the mission

Run the mission.

Before you can enter one of the aircraft, a lot of information runs down the right side of the screen:

```
+++debug: changed generic 'inf' from <Soldier M4> to <Soldier M4 GRG>
*** monitoring events defined in <Events to monitor>:
monitoring event <S_EVENT_SHOT = 1>
monitoring event <S_EVENT_HIT = 2>
monitoring event <S_EVENT_TAKEOFF = 3>
monitoring event <S_EVENT_LAND = 4>
monitoring event <S_EVENT_CRASH = 5>
monitoring event <S_EVENT_EJECTION = 6>
monitoring event <S_EVENT_UNIT_LOST = 30>
monitoring event <S_EVENT_LANDING_AFTER_EJECTION = 31>
monitoring event <S_EVENT_PARATROOPER_LANDING = 32>
monitoring event <S_EVENT_DISCARD_CHAIR_AFTER_EJECTION = 33>
monitoring event <S_EVENT_BIRTH = 15>
```

Take note, and enter “Frog One”.

2.9.2 In-Mission

As soon as you enter the cockpit, a new message appears:

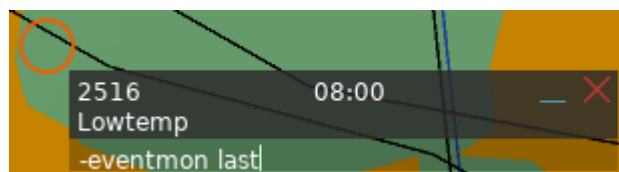
```
*** event <S_EVENT_BIRTH = 15> for player = Lowtemp in unit <Frog One>
```

Since we are currently monitoring event ID 15 (= “Birth” event”) and your player Frogfoot was ‘birthed’ (spawned) into the game, DCS invokes the event, and The Debugger alerts you to that fact, with some more information added as QoL.

Since we are interested in this event (I want to find out what that event also tells us), go to F-10 Map view, activate the Mark Label tool, and click anywhere in the map. Invoke

```
-eventmon last
```

to get an analysis of the last event that The Debugger recorded for us:



Now, unless you are (like me) afflicted with the incurable disease coding addiction, the result from The Debugger is some meaningless gibberish that you can safely ignore. If you are into mission scripting, however, take note that The Debugger can give you a detailed analysis of what the event table contained: ID = 15, time = 0 (at the very beginning), and an initiator that is defined (but unresolved in the table: we only have the ID)

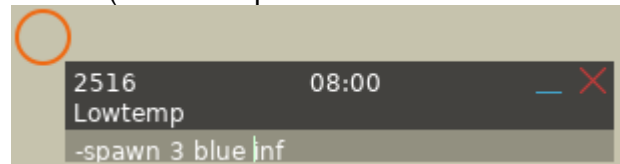
```
analysis of <event>
===
event: [
  id: 15
  time: 0
  initiator: [
    id: 16777728
  ] - end initiator
] - end event
=== analysis end
```

Now zoom out. Note the A-10A on approach to Kolkhi. It will be a while, but it's currently tasked with landing at Senaki-Kolkhi's 09. Locate your own aircraft on the map, and zoom into it.

Next, we are going to try out The Debugger's "spawn" ability. Click on the Mark Label icon, and then place it close to and in front of your aircraft (for the simple reason that we can see what we spawn from the cockpit). Issue

```
-spawn 3 blue inf
```

and click outside the text box to activate. If everything goes well, three new Soldiers (the Soldier M4 GRG type, the newer Soldier M4 kind with better animations) belonging to blue appear on the map close to the location where you placed the Mark Label.

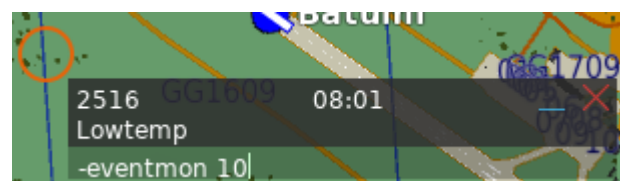


Note also that The Debugger reports that three "Birth" events (ID = 15) have occurred:

```
*** event <S_EVENT_BIRTH = 15> for unit <Soldier M4 GRG-76544-1>
*** event <S_EVENT_BIRTH = 15> for unit <Soldier M4 GRG-76544-2>
*** event <S_EVENT_BIRTH = 15> for unit <Soldier M4 GRG-76544-3>
```

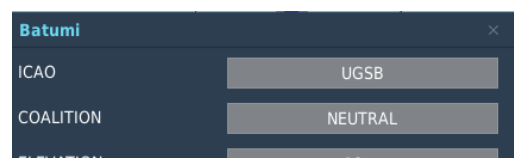
This of course corresponds to the three infantry units that we spawned. Since 15 (Birth event" is among the events that

Next, we add the "Airfield Captured" event to the events that we want to observe:

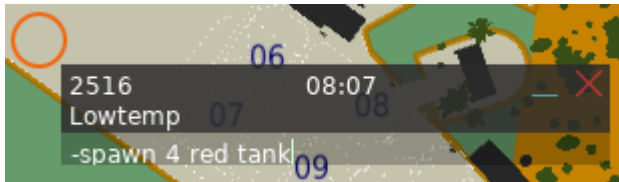


```
*** eventmon: added event <S_EVENT_BASE CAPTURED = 10>
```

Now, pan the map to Batumi, zoom in, and click onto the airfield. A window comes up that gives us a lot of information about Batumi. Most importantly, it tells us that Batumi currently is NEUTRAL



Airfields change their affiliation when one faction manages to be the only faction with ground units inside a 2km radius of the airfield. Let's capture Batumi for RED by placing 4 tanks onto the airfield.



Once we click outside, we receive four “Birth” events (the four tanks that we spawned), quickly followed by The Debugger noting that

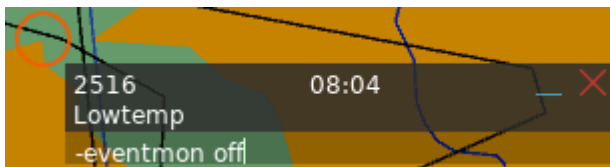
```
*** event <S_EVENT_BASE_CAPTURED = 10> for unit <T-90-76544-4>
```

If we now click on Batumi again, we find that Batumi belongs to the RED faction.

Take another look at the A-10A on approach in Senaki. Wait until it lands (some two minutes after the mission starts)

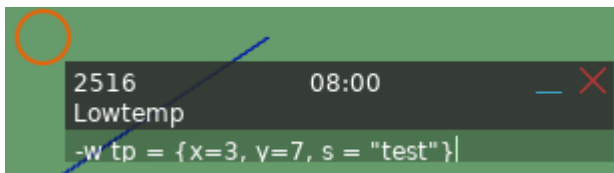
```
*** event <S_EVENT_LAND = 4> for unit <Lander>
```

Now clear all events from being reported.



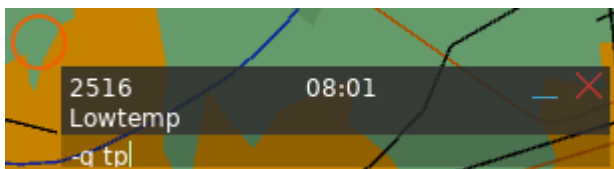
```
*** eventmon: removed all events from monitor list
```

And now for some rather arcane stuff that may be of questionable use for 99.9% of all DML users, and merely serves to show just what The Debugger can do:



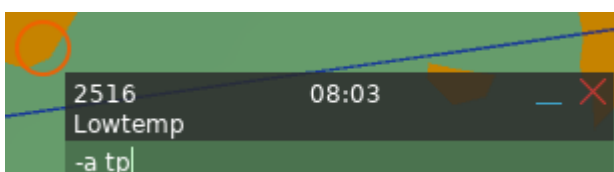
```
[08:01:06] <tp> set to <{x=3, y=7, s = "test"}>
```

The Debugger has now created (or replaced) a new Lua table “tp” in the Mission Scripting Environment. Proof:



```
[08:01:59] <tp> = [Lua Table]
```

And should we want to analyze the table “tp” we get



```
[08:03:38] <tp> = analysis of <tp>  
===  
tp: [  
  y: 7  
  x: 3  
  s: test  
] - end tp  
=== analysis end
```

So, yeah, The Debugger really can do these things.