

cf/x

# Dynamic Mission Library for DCS

© 2022 by Christian Franz and cf/x

Version 0.95 – 2022-01-20

# Table of Contents

1	Welcome – What is this?.....	8
1.1	About this Document.....	11
1.2	Part IV - Demo Missions: Have a look!.....	12
1.3	What's up with those “(Lua Only)” remarks? .....	12
1.4	DISCLAIMER.....	12
1.5	Copyright .....	13
1.6	Acknowledgements.....	13
2	Introduction.....	14
2.1	Functional Overview .....	16
2.1.1	Zone-based Enhancements for ME.....	16
2.1.2	Feature Enhancements.....	19
2.1.3	Foundation Level (Lua Only).....	21
2.2	Architecture (Lua Only) .....	24
3	Using DML.....	26
3.1	How to import Modules into a Mission.....	26
3.2	Important Concepts.....	28
3.2.1	Zones and Attributes.....	28
3.2.2	ME Flag integration into DML.....	29
3.2.3	Module Configuration Zones .....	30
3.2.4	Orders.....	30
3.2.5	Spawn Formations .....	32
3.2.6	Spawning: Type String and Type String Arrays.....	33
3.2.7	Ownership / Owned Zones.....	34
3.3	DML Mission Design Philosophy (Lua Only) .....	35
3.3.1	Game States.....	35
3.3.2	Update() Loop.....	35
3.3.3	Event Handler .....	36
3.3.4	ME-provided “Configuration Zones” .....	37
3.3.5	The Start() Method.....	38
3.3.6	The “Main” Skeleton.....	39
3.4	Using Zone Enhancements.....	41
3.4.1	All Zone Enhancements.....	42
3.4.2	cfxSmokeZones .....	43
3.4.3	cfxObjectDestructDetector .....	45
3.4.4	cfxSpawnZones .....	48

3.4.5	cfxObjectSpawnZone.....	53
3.4.6	cfxCargoReceiverZone .....	58
3.4.7	cfxArtilleryZones .....	60
3.4.8	cfxOwnedZones.....	64
3.4.9	FARP Zones .....	69
3.4.10	cfxMapMarkers .....	71
3.4.11	cfxNDB .....	72
3.5	Using Stand-Alone Features .....	75
3.5.1	Player Score .....	77
3.5.2	cfxHeloTroops.....	80
3.5.3	jtacGrpGUI.....	83
3.5.4	csarManager.....	84
3.5.5	Limited Airframes (tbc).....	88
3.5.6	Guardian Angel.....	89
3.5.7	parashoo.....	93
3.5.8	Civ Air .....	94
3.5.9	Artillery UI .....	95
3.5.10	Recon Mode .....	98
3.5.11	ssbClient.....	103
3.5.12	ssbSingleUse (tbc).....	107
3.5.13	cfxMon Development Tool (Lua Only) .....	108
3.5.14	Module Name .....	110
3.6	Using Foundation (Lua Only) .....	111
3.6.1	dcsCommon (Lua only).....	111
3.6.2	cfxPlayer (Lua Only) .....	115
3.6.3	cfxZones (Lua Only).....	121
3.6.4	cfxCommander (Lua Only) .....	125
3.6.5	nameStats (Lua Only) .....	131
3.6.6	cargoSuper (Lua Only).....	135
3.6.7	cargoManager (Lua Only) .....	140
3.6.8	cfxGroundTroops (Lua Only).....	143
3.6.9	cfxGroups (Lua Only).....	149
4	Foundation API .....	151
4.1	dcsCommon API.....	152
4.1.1	Miscellaneous Methods .....	152
4.1.2	Table / String Managements .....	155
4.1.3	Vector Math .....	157

4.1.4	Airfield, Landable Ships and FARP .....	158
4.1.5	Group handling .....	160
4.1.6	Unit Handling .....	161
4.1.7	Spawning Units / Group, Routes, Tasks.....	163
4.1.8	Static Objects.....	166
4.1.9	Coalition.....	166
4.1.10	Event Handling .....	167
4.2	cfxPlayer API .....	168
4.2.1	Tables.....	168
4.2.2	Callback Handling .....	168
4.3	cfxZones API .....	170
4.3.1	Testing.....	171
4.3.2	Management.....	173
4.3.3	Properties .....	175
4.3.4	Spawning.....	178
4.3.5	Miscellaneous .....	178
5	Tutorial / Demo missions.....	180
5.1	Overview.....	180
5.2	Smoke'em! DML Intro.miz.....	183
5.2.1	Demonstration Goals .....	183
5.2.2	What To Explore .....	183
5.2.3	Discussion .....	184
5.3	Object Destruct Detection (ME Integration).miz .....	185
5.3.1	Demonstration Goals .....	185
5.3.2	What To Explore .....	185
5.3.3	Discussion .....	186
5.4	ADF and NDB Fun.miz (tbc) .....	187
5.4.1	Demonstration Goals .....	187
5.4.2	What To Explore .....	187
5.4.3	Discussion .....	187
5.5	Artillery Zones Triggered.miz .....	188
5.5.1	Demonstration Goals .....	188
5.5.2	What To Explore .....	188
5.5.3	Discussion .....	190
5.6	ME Triggered Spawns.miz .....	191
5.6.1	Demonstration Goals .....	191
5.6.2	What To Explore .....	191

5.6.3	Discussion .....	192
5.7	Spawn Zones (training and lasing).miz .....	193
5.7.1	Demonstration Goals .....	193
5.7.2	What To Explore .....	193
5.7.3	Discussion .....	194
5.8	Moving Spawners.miz .....	196
5.8.1	Demonstration Goals .....	196
5.8.2	What To Explore .....	196
5.8.3	Discussion .....	196
5.9	Helo Trooper.miz .....	198
5.9.1	Demonstration Goals .....	198
5.9.2	What To Explore .....	198
5.9.3	Discussion .....	200
5.10	Helo Cargo.miz .....	202
5.10.1	Demonstration Goals .....	202
5.10.2	What To Explore .....	202
5.10.3	Discussion .....	203
5.11	Artillery with UI.miz .....	205
5.11.1	Demonstration Goals .....	205
5.11.2	What To Explore .....	205
5.11.3	Discussion .....	207
5.12	Missile Evasion (Guardian Angel).miz .....	209
5.12.1	Demonstration Goals .....	209
5.12.2	What To Explore .....	209
5.12.3	Discussion .....	209
5.13	Recon Mode.miz .....	211
5.13.1	Demonstration Goals .....	211
5.13.2	What To Explore .....	211
5.13.3	Discussion .....	212
5.14	Owned Zones ME Integration.miz .....	213
5.14.1	Demonstration Goals .....	213
5.14.2	What To Explore .....	213
5.14.3	Discussion .....	214
5.15	Keeping Score.miz .....	215
5.15.1	Demonstration Goals .....	215
5.15.2	What To Explore .....	215
5.15.3	Discussion .....	216

5.16	DML Mission Template.miz – (Lua Only)	217
5.16.1	Demonstration Goals	217
5.16.2	What to explore	217
5.16.3	Discussion	218
5.17	Landing Counter.miz – (Lua Only)	222
5.17.1	Demonstration Goals	222
5.17.2	What To Explore	222
5.17.3	Discussion	222
5.18	Event Monitor.miz	224
5.18.1	Demonstration Goals	224
5.18.2	What To Explore	224
5.18.3	Discussion	224
5.19	Mission.miz	225
5.19.1	Demonstration Goals	225
5.19.2	What To Explore	225
5.19.3	Discussion	225
5.20	Mission.miz	225
5.20.1	Demonstration Goals	225
5.20.2	What To Explore	225
5.20.3	Discussion	225



cf/x  
Dynamic Mission Library  
for DCS

# PART I: INTRODUCTION & OVERVIEW

# 1 Welcome – What is this?

Welcome to this document – and thank you for taking the time to RTFM – you are very wise indeed to read this, as much of what’s written here should make using DML more enjoyable for you and help shorten the time it takes to use DCS Dynamic Mission Library (DML) in your own missions.

So, what is DML? It’s a **mission-building toolbox that does not require Lua**, yet also provides comprehensive support if you do want to use Lua. At its heart are modules that **attach themselves to Mission Editor’s (ME) Trigger Zones** to provide new abilities. Mission designers control abilities in ME by adding ‘Attributes’ to these Trigger Zone.





Name	Value	
NDB	121.5	
soundFile	distressbeacon.ogg	

For example, when you add above attributes to a trigger zone, the “cfxNDB” module automatically activates for this zone, and starts an NDB at the zone’s center at 121.5 MHz, playing the “distressbeacon.ogg” sound file on that frequency.

Through this simple mechanism, adding complex new abilities to missions becomes a snap (or, at least, much easier). Since **you control DML from inside ME**, you do not have to mess around with Lua scripts – all DML modules take their run-time data from Trigger Zone attributes. You edit those in ME: Trigger Zones already have attributes, editing them is built into ME. If you have ever created a Trigger Zone, you have already seen ME’s zone attributes. You likely ignored them because they have had little practical use. Until now. We’ll use zone attributes to put DCS mission creation into super-cruise.

**DML can reduce advanced tasks (such as adding CSAR missions) to placing trigger zones and adding attributes.**

If that isn’t enough, DML modules **can be triggered with ME flags**, while **others can set ME Flags** when they activate. For example, spawn zones can be instructed to watch flag 100, and spawn every time when that flag changes its value. Other modules can be told to increase a Flag (e.g., 110) every time they activate. This allows you to integrate the modules in your normal ME mission design workflow without having to resorting to outside means.

Name	Value	
f+1	110	
f=0	100	
f=1	200	
f-1	210	

If a module requires configuration data, it starts up with default values, and then looks for a – surprise! – Trigger Zone that might contain the attributes that you want to change for this mission. **You can configure your modules from within ME** – you don’t have to change a single line of code.

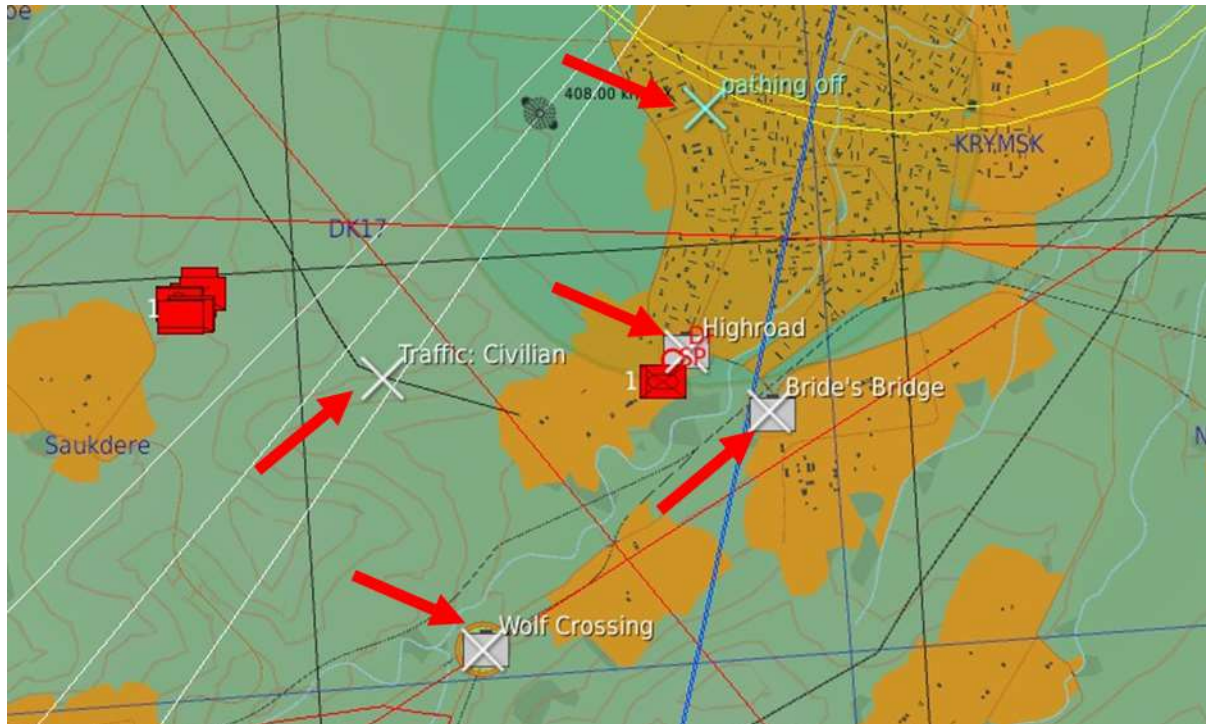


DML has something in store for every mission designer – novices and veterans alike. And for mission designers who have discovered Lua, DML can super-charge their abilities. That being said, **Lua knowledge is not required** to use DML in your missions. At all.



This would be the perfect moment to try the first of the many demos that come with DML – so if you want to know what this is all about, just go here (→ [Smoke'em! DML Intro.miz](#)) and find out!

Let us look at a real-life DML-enhanced mission:



Note the five Trigger Zones on the map (follow the unobtrusive red arrows). As mentioned, DML uses ME Trigger Zones and attaches its own modules to them. That way, mission designers can simply place new functionality by adding standard Trigger Zones to the map - without requiring any Lua. You then add a few attributes to the Trigger Zone, and DML's modules home in on them automatically.

Name	Value	
pathing	offroad	

Above screenshot was taken from my "[Integrated Warfare: Pushback](#)", a mission that uses DML to dynamically create ground forces and that require the player's air support to win. On the map, I placed various zones to

- Add conquerable zones ("Wolf Crossing", "Bride's Bridge", "Highroad") – these are zones that, when captured by blue or red, automatically produce ground forces that defend the zone against invaders and seek out and capture other conquerable zones
- Control civilian air traffic ("Traffic: Civilian")
- Control AI's pathing for ground forces ("pathing off")

All zones use simple, ME editable attributes (like "pathing", "offroad") to tell DML what to do. In the end, writing such a mission amounts to just a little more than placing zones and adding attributes. After all, the trick is coming up with a good mission idea – putting it together should be easy. With DML it may have become a bit easier.

Behind the scenes, DML also provides a collection of **Foundation** modules that lack ME integration. Using these modules directly is not intended for beginners and requires a modicum of Lua-knowledge; they provide ready-made, tested, convenient access to many

functions that mission designers would traditionally code by themselves (or use ready-made libraries).

So, what's in DML right now? In a nutshell here's what you get:

- **Drop-in Modules (no Lua knowledge required)** that add complete functionality to a mission – for example
  - CSAR Missions
  - Limited number of pilots (ties in with CSAR Missions)
  - Civilian Air traffic
  - Automatic Recon Mode
  - Slot Blocking Client (SSB based) for preventing spawns on enemy airfields
  - Protection from missiles
  - Helicopter Troop Pick-up, Transport and Deployment
  - Score Keeping
- **Zone Enhancements** that interactively **attach new functionality to Zones in ME (no Lua required)**. They provide diverse functionality such as
  - Dynamic Ground Troop Spawning
  - Dynamic Object/Cargo Spawning
  - Artillery Target Zones
  - Conquerable Zones and FARPS
  - (moving) NDB
  - Map/Scenery object destruction detector
- **Foundation**, a library of ready-to-use methods (**only for mission designers who use Lua**). They support
  - Advanced Event Handlers for mission and player events
  - Zone management and attaching/reading zones attributes
  - Inventory keeping
  - Managing orders and pathing for troops
- **Multi-player supported out-of-the-box**. All modules work for single- and multiplayer missions, including modules with user interaction via communications.
- **A collection of fully documented Tutorials / Demos** that serve to illustrate how the more salient points of DML can be used to quickly create great mission. They aren't flashy. They hopefully are helpful instead.
- **A hefty Manual** that I can lord over you and yell "RTFM" whenever you have a question. Yup, that's definitely why I wrote it.

Of course, this is just the beginning – DML is far from complete, and there are lots of new avenues to explore. Based on feedback, I expect DML to evolve, and to add new and exciting abilities. Until then, I hope that you enjoy the ride!

-ch

## 1.1 About this Document

This document is divided into multiple parts:

- **Part I: Introduction & Overview (you are reading this part right now)**

gives you a bird's eye view of the library: how the various parts fit together, and what they are designed for. Read this first, as having a rough sketch of the map often helps understanding the details. Because – when the part that you are reading refers to something that will come later, knowledge about where that part belongs to makes it much easier to keep calm and carry on reading

- **Part II: Using DML**

This is the heart of DML's documentation, and we cover a lot of ground here:

- We begin with some important DML concepts (e.g., Zone Attributes, ME Integration, ...), and how the modules work together in general. Reading this part is very important, since it helps to understand why modules are built the way they are.
- We then go through every module and take a closer look: what it does, how to use it in your mission, and (for those who are interested) walk through the API (if one exists).  
You can skip any module that doesn't interest you; all module descriptions are structured identically; they can be read in any order. Moreover, the Foundation modules are inherently technical, and should only be read by people who absolutely, positively want to get more out of DML by using Lua.
- Note that if a module's section in this part is headed with "Lua Only" (e.g., "dcsCommon (Lua Only)"), you can safely ignore that section until the time that you want to light the Lua 'burner. Understanding such a module is only required for advanced DML uses.

- **Part III: Foundation Reference**

This part provides a comprehensive reference to Foundation's Lua API that we left out of Part II.

Neither part is meant to stand on its own; part I is too short to convey much helpful information, while part II and III provide an ocean of detail that you can drown in. So read about what's in store in part I, and then embark to navigate parts II and III. When you get lost in the details, make sure to refer back to part I, get your bearings, and then head out back on track.

And – explore! There are demo missions to investigate and play around with – most are documented in Part IV: Tutorials / Demo. These missions are meant to be taken apart, dissected, and modified. When in doubt, load up ME, and experiment. Many questions are much easier answered by equal amounts of experiments and creativity. Finding out that something happens in a certain way is often as much fun as finding out why.

## 1.2 Part IV - Demo Missions: Have a look!

DML comes with a set of demo missions that are designed to illustrate some of its major abilities and provide a reference to how you can use them in your own mission. With very few exceptions, these demos require no Lua knowledge.

Note that the demo missions, from a player's perspective, are tepid at best: there's very little pizzazz in evidence when you play them. As mission designer, however, they may very well knock your socks off – when you realize how little effort it takes you to add these features to your next mission. And be sure to read each demo mission's Discussion section, as you may have missed the best.

Examining the demo missions can help jump-starting your own mission designs with DML - as many demos either focus on a module's features, or how these modules work together. When you are done putting the mission through its paces, read the 'Discussion' part of each demo again to find out about some interesting point you probably have overlooked.

Finally, **if you are interested in starting mission scripting with Lua**, you will find that **there are a couple of Lua-specific Tutorials** that could be very interesting, as they illustrate **how to structure an effective mission management script in Lua** (expanding on →DML Mission Design Philosophy (Lua Only)).

## 1.3 What's up with those “(Lua Only)” remarks?

DML has many uses. It provides strong ME integration, and **most modules can be used without ever having to write a single line of code.**

That being said, DML can truly supercharge your missions – provided you know how to write Lua code. DML provides strong API for those of you who are bold enough to venture into those regions where “there be dragons”: the Lua Scripting Abyssal. While ME does a terrific job in harnessing many of DCS's mission capabilities, it can but scratch the surface of the real underlying mission engine. Lua scripts can; Lua scripts can go *deep*. They regularly catch fish no ME-only designed mission could even dream of.

But that ride isn't free: coding is not for everyone; it's a disposition as much as an aptitude. So, if you don't hear the permanent siren song with which words like *invocation*, *class* or *callback* are constantly calling out to coders, do not fret: You aren't thusly afflicted. **Simply know that you can safely skip any chapter that I labelled “(Lua Only)”.**

And to you, the coder I say: “Suit Up!”

## 1.4 DISCLAIMER

Let's make this short – we are not lawyers. Understand that by doing anything that is described, recommended, suggested, alluded to, inferred, or merely hinted at in this document, you may cause incredible damage, cause war, and maybe even end life as we know it.

**By using DML you accept and irrevocably commit to not holding me, Christian Franz, nor anyone I know, did know, or might know accountable for anything that results from using any part of DML and/or associated materials.**

You have been duly warned, and you will not try to indemnify anyone but yourself for any damages resulting from anything that involves DML.

## 1.5 Copyright

This document, and all accompanying code and demos are copyright © 2021, 2022 by Christian Franz and cf/x AG. **You are free to use DML for any non-commercial purpose,** provided you include an attribution

“Uses DML © 2022 by cf/x and Christian Franz”

in that work’s documentation.

For commercial use, please contact me.

## 1.6 Acknowledgements

(tbd)

## 2 Introduction

Dynamic Mission Library (DML) is a collection of modules that enhance a mission author's scripting capabilities for Eagle Dynamic's DCS World Mission Editor. It integrates new abilities like unit spawners, artillery, civilian air traffic and CSAR missions into Mission Editor.

DML comprises of:

- **Zone based Enhancements for ME**

These enhancements use Mission Editor's Trigger Zones to attach ("anchor") themselves, and "Attributes" to control these new features. In other words, Zones tell DML "where", and their attributes "what": attributes tell "Spawn Zones" what units to spawn and when, they describe to "Map Markers" what to write onto the F10-Map, or control how "Civ Air" lets air traffic flow.

To "anchor" a module to a Trigger Zone, Mission Designers simply add an attribute – DML then automatically attaches the correct module to that zone (see the documentation to find out what attributes control). Zone Enhancements can "stack", meaning that you can anchor multiple modules to the same zone, e.g. Map Marker, CSARBASE and FARP can all attach to the same zone.

Using Trigger Zones with attributes has numerous enormous advantages:

- No messing around with scripts
- Mission Designers can use ME's visual editing tools to place functionality
- we separate a module's functionality (which is code-based, and should not concern designers) from visual mission building
- we have a graphical representation (the enhanced zones) of where we place the new functionality
- we can use copy/paste to quickly replicate enhanced zones over the map,

which conveniently integrates DML into your normal mission editing workflow

- **Full-Feature Modules**

These are ready-made, drop-in packets that add new features like CSAR (Combat Search And Rescue) missions simply by adding them to a mission.

- **Zone-based configuration and data access**

Modules use Trigger Zones to provide configuration details and mission data to modules. That way you can change or configure a module without having to access or change the underlying code.

- **Functional Libraries (Foundation, Lua-Only)**

These Lua-only modules provide ready-made methods for advanced mission authors. Unlike other modules they require Lua know-how.

- **Demo Missions**

Often, a picture is worth a thousand words. And a demo mission is worth ten tutorial videos. DML comes with a pack of missions that are curtailed to demonstrate (as opposed to 'show off') DML's capabilities, and how to integrate them from a mission designer's standpoint. They are short on sugar, and high on nutritional value. And they come with a dedicated part in this documentation, so be sure to walk through

each demo with the documentation in hand, or you may miss some of the finer points.

**All DML modules are lightweight and have negligible performance impact;** the entire library is self-contained. No other libraries (e.g., 'MIST' or 'MOOSE') are required; there are no known conflicts when you run other libraries side-by-side with DML.

## 2.1 Functional Overview

To use, DML requires that you are able to perform three basic steps within DCS Mission Editor (ME): create actions, place Trigger Zones, and add “attributes” to Trigger Zones. Beyond that are no requirements; **DML does not require any Lua knowledge** from mission designers.

So let's take a tour through DML, and just briefly stop at the main attractions. Part II will go into detail, for now let's get the Big Picture: how does DML work with ME, and what does it offer a mission designer?

### 2.1.1 Zone-based Enhancements for ME

These are modules that everyone can use from within ME without writing any code. All that is required is that mission designers place Trigger Zones, and then use ME to add “Attributes” to these zones.

An “Attribute” (also called “Property”) is a Name/Value pair (example: “Pilot/Iceman”) that you add to a Trigger Zone with ME's Trigger Zone editor (see right)

All Zone-based enhancements use this simple mechanism to pass information and control its abilities.

To add an Attribute, inspect a Trigger Zone in ME. Then click on the “Add” button, and edit the “Name” and “Value” fields. To change an attribute's name or value, click into the fields. If you leave a Value empty, the Attribute itself remains in existence and retains the value <empty>. When encountering an empty attribute, all modules simply use a default value instead – they do the same when an attribute is absent)

To find out which attribute names and values are defined, please see the relevant sections of this documentation.

DML already supports a large and varied host of enhancements that all use Zones and Attributes to control their features. Below, please find a short overview of what DML has on offer:



#### 2.1.1.1 Standard cfxZones attributes

DML uses standard DCS (Mission Editor) zones as “anchor” for its modules. Any zone that is placed with ME is automatically processed by DML and passed to its modules. In addition to simply mapping out a region on the map, zones managed with DML understand attributes (see above). When present, certain attributes mark a Zone as an anchor for a specific



module. For example, any zone that has the 'smoke' attribute will automatically anchor 'cfxSmokeZone' functionality (see below)

Zones can serve as anchor for multiple modules at once, although this may be limited by DCS itself: if you define a zone to be an anchor for both a smoke zone (see below) and a moving zone, the *zone* moves as you intent; the smoke stays in place for 5 minutes, and then jumps to wherever that zone has moved in the intervening time; the smoke then again remains in place for another 5 minutes, while the zone itself may move on to another place.

When you use zone based enhancements, **all zones**, regardless of other modules they may anchor, **support a number of attributes** in DML:

- **linkedUnit**  
Makes the zone move with the indicated unit. Works only inside the logical confines of DML, not supported by ME
- **useOffset**  
When using "linked Unit" (see above) maintains the spatial offset between the zone and the linked unit
- **owner**  
Assigns an owner (red, blue, neutral) to this zone. Only available within DML

#### 2.1.1.2 *cfxSmokeZones*

Add a **permanent, colored smoke effect** to the center of the zone. It doesn't stop smoking for the entire mission. You control smoke color with the 'smoke' attribute's value. Compatible with other zone extensions.

#### 2.1.1.3 *cfxObjectDestructDetector*

This little gem's goal is to greatly simplify detection of when a **Map Object is destroyed** – be it a bridge or building. It **tightly ties into ME's** ability to assign a zone to a building or other object and provides mission designers with simple functionality to directly manipulate flags.

More advanced scripter can take advantage of the callback functionality that is invoked when the marked structure is destroyed.

#### 2.1.1.4 *cfxSpawnZones*

This adds the ability to **spawn troops** in a zone - automatically, and on demand. Spawns can occur once and multiple times. The spawned troops can receive complex orders.

#### 2.1.1.5 *cfxObjectSpawnZones*

Very similar to *cfxSpawnZones*, this zone **spawns cargo and "static" (scenery) objects** instead of combat units. Since a peculiarity of DCS is that helicopter cargo items are static objects, you can use this enhancement to conveniently spawn cargo for helicopters to sling-load.

Object spawn zones can be linked to ships and therefore be used to spawn static objects there (make sure to use `Offset` to make objects spawn relative to the ship's center point)

#### *2.1.1.6 cfxOwnedZones*

This adds the ability of creating **zones** (areas on the Map) **that can be captured**. They currently also offer some specialized spawning abilities, depending on which faction holds the zone. They can also be marked by owner on the F10 in-game map

#### *2.1.1.7 cfxArtilleryZones*

This simulates artillery target zones for interaction with FO. Can **simulate artillery bombing**. Later versions may connect to artillery units in range.

#### *2.1.1.8 cfxCargoReceiver*

These specialized zones work in conjunction with the `cfxCargoManager` module. **Delivering** (unhooking) a helicopter's **sling-loaded cargo** in such a zone **sets ME flags** and generates events that other scripts can subscribe to. Cargo Receiver also provides text-based messages for helicopter pilots that approach them with information aimed to **guide them to the drop zone** (once the helicopter is close enough).

#### *2.1.1.9 FARPZones*

Adds the ability to **make FARPs conquerable** like Owned Zones, with easy placements of defenses, and ensures that all resources for reload and repair are available at start and after capturing a FARP. Unlike Owned zones, ownership is managed by the FARP that must be inside the FARP Zone. Since Owned Zones and DCS FARP follow the same rules for ownership, there is seldom a conflict; the main difference between an Owned Zone and a FARP Zone is that `GroundTroops` (the module that manages ordered troops) is unable to resolve a troop pileup in a FARP. If this situation arises, players must eliminate enemy troops themselves.

#### *2.1.1.10 cfxMapMarkers*

This small script adds the ability to **place arbitrary map notes** (text) on the F-10 in-game map, visible to either faction, or all factions. The text appears wherever the mission designer places the zone

## 2.1.2 Feature Enhancements

These enhancements add 'drop-in' functionality to DCS missions. They, too, can be customized by adding attributes in ME; some interact with, or expand the capabilities of existing Zone Enhancers (CSAR Manager, Helo Troops, Player Score).

### 2.1.2.1 *Player Score / Player Score UI*

Provides **simple score-keeping** and **kill-tabulating**, fully MP-capable, based on *player name* (not unit). Supports individual "named" unit score (i.e. a special score of 100 for the unit with name "Theater Commander") and type scores (e.g. a score of 20 for all units of type "BTR-80"). Has a ready-made, MP-capable UI

### 2.1.2.2 *Recon Mode*

A module that allows planes (AI and Player) to **automatically record enemy groups on the F10 map** for all players on the same side to see. Supports priority- and black-listed groups.

### 2.1.2.3 *Civ Air*

This module provides **AI-controlled civilian (well, neutral) air traffic** that flies between airfields in the region. Current version requires some customization for other maps than Caucasus.

### 2.1.2.4 *Helo Troops*

A drop-in feature to enable **player-controlled troop helicopters** (Hind, Hip, Huey) to pick up and deploy infantry. Can interact with spawn zones to request troop production.

### 2.1.2.5 *CSAR Manager*

A drop-in feature that **provides CSAR Mission** support: pick up downed pilots and deliver them to safe zones. Functions out of the box; requires the author to add safe zones with ME to designate the place where rescued personnel can be delivered. Additionally, it provides a convenient and easy ME interface to instantly create CSAR missions upon mission start.

### 2.1.2.6 *Artillery UI*

Provides an **interface for Artillery Zones**. Allows helicopters to call in smoke to artillery zones, and when close enough and in direct line of sight (LOS) to the zone's center, order artillery to fire.

### 2.1.2.7 *Limited Airframes*

This module provides two significant additions to any mission that it is added to:

- **Limits the number of pilots** ("airframes" since each time you lose an airframe you lose a pilot) per side. So even if a mission allows for a multitude of airframes to choose from, this module limits the number of "lives" a side has until the mission is lost

- To offset the pilots lost, this module **automatically interfaces with the CSAR Manager** module (if present) to generate CSAR missions for ejected player, so helicopter pilots can attempt to retrieve a downed pilot (at the risk of another pilot).

#### *2.1.2.8 Guardian Angel*

A module that **destroys missiles inbound** on certain airframes just before they hit. Not 100% safe, but very close. Will give statistics about missiles dodged. Can be used to simulate 'jamming' of missiles, and is mainly intended for missile evasion training purposes.

#### *2.1.2.9 parashoo*

A small module that removes parachutists once they reach the ground. Its main benefit is that it declutters a player's F10 map (i.e., it avoids too many parachute icons) in long-running missions.

#### *2.1.2.10 Recon Mode*

Allows reconnaissance flight (player and AI) with automatic marks on the F10 in-game maps. Supports blacklist (groups that are never found) and priority target lists.

#### *2.1.2.11 ssbClient*

A module that allows **slot-blocking** for **aircrafts** on airfields that currently **do not belong** to the aircraft's **faction**. Requires that the server (only the server) that is hosting the mission has the SSB script running.

#### *2.1.2.12 ssbSingleUse*

A module that allows **slot-blocking for aircrafts** that have previously **crashed**. Requires that the server (only the server) that is hosting the mission has the SSB loaded and that SSB's kickReset option is turned off (set to false)

#### *2.1.2.13 cfxmon*

Highly specific **debugging tool** that allows mission designers to **monitor** every **callback** provided by DML. Callbacks can be selectively disabled.

### 2.1.3 Foundation Level (Lua Only)

These are modules that provide methods that accomplish common mission tasks: calculating distances, issuing orders, creating “events”. All modules further down in the architecture require at least one (often more) of these modules. **If you do not intend to write your own Lua scripts that tap into DML, you can safely skip this section.**

#### 2.1.3.1 *dcsCommon*

This is DML’s Bedrock. All other libraries and enhancements require this **collection of common methods**. Look at the API description to find out what is provided. Generally: if it’s something basic/common you want to do, there’s a method for it in *dcsCommon*, especially if you need access to a unit’s basic information like heading, speed, or position in relation to another unit or aerodrome. It provides an improved notification/callback method for you that allows you to filter, pre- and post-process events in a much more friendly way than DCS does.

*dcsCommon* also provides many of the “primitives” you can use to assemble and spawn groups – however, like with other foundation methods, other modules that are higher up in the architecture usually provide more powerful functionality.

Finally, *dcsCommon* provides one central pillars for mission designers who create their missions based on **update/event** cycles (with *cfxPlayer* providing another: player events)

#### 2.1.3.2 *cfxZones*

This is a collection of methods that specialize in **handling DCS Zones** and providing easy **access to properties**. When scripting with DML, script authors should always use *cfxZones* instead of trying to access DCS’s mission zones directly. This library provides support for reading attributes, handles **moving zones** (called ‘linked zones’ in *cfx* parlance), and is the main building block for all Zone-based enhancements. Using *cfxZones* in your own code makes mission scripting with zones a lot easier. *cfxZones* implicitly adds new attributes (like ‘owner’) to zones so that these attributes are always available to scripts. Mission designers can override these implicit attributes simply by adding it explicitly to a zone.

Mission authors usually only utilize a few of this module’s methods directly, utilizing modules/callbacks provided by modules higher up in the architecture instead.

#### 2.1.3.3 *cfxPlayer*

This library mainly **provides callback functionality for player events**, and manages updating player information transparently. It provides a convenient callback framework so mission scripts can easily keep up to date with all player information. *cfxPlayer* is most useful when scripts need to implement multi-player functionality or a GUI (e.g. via Communication) that must differentiate between player-controlled Units/Groups. Since most Feature Enhancement modules are multi-player enabled, they require this module even if the mission is intended for single-player use only.

Interacting with this module is mainly via subscribing to player events and then writing code that handles whatever needs to be done when those events happen.

#### 2.1.3.4 *cfxCommander*

This small library provides functionality to **issue orders to groups** via a group's controller. Emphasis is here on providing simple methods to schedule orders; this is important because ordering units immediately after they have been created can cause Issues, and orders are often given in a sequence (stop now, then start moving in 10 seconds). It is a purely convenience library that provides scripting shortcuts.

One more advanced feature that this module brings to the table is for **pathing** in conjunction with pathing attributes and pathing zones on the map which allow the designer to optimize unit pathing (groups can automatically follow roads and drive off-road in certain areas that are designated by zones).

#### 2.1.3.5 *cfxGroundTroops*

This is the hub module that provides convenient “**Orders**” for **DCS groups of ground units**. It manages a pool of Troops – DML parlance for a DCS Group with “orders”. Scripts interact with this module mainly by configuring the task loop at the beginning, and later by adding troop tables to, and removing them from the pool it manages.

Troops added to the GroundTroops pools all have orders (see the section on ‘Orders’ later in this document) and they are managed in-game by GroundTroops’ task management loop automatically.

Ground Troops **handles group** tasking via **orders, route them**, can autonomously **change a group's orders** if need arises, automatically **lase targets**, and remove troops from the management queue when they get stuck. Groups in GroundTroops management pool that are **destroyed are automatically removed** from the queue, and GroundTroops understands multiple tasking loop methods for enhanced performance: to manage performance, GroundTroops can put new units into order queues and dynamically handles adding them to the task loop. It supports multiple tasking models (all at once, sequential) to fine-tune performance. It can also **resolve ‘pile-ups’** inside owned zones, when multiple enemy groups mutually block each other, and neither can clearly resolve capture.

Many Zone Enhancers automatically submit groups to the GroundTroops module for minimal interaction. Others can remove them from the pools and later return them, and even understand transforming orders.

#### 2.1.3.6 *nameStats*

This is a module that provide an **easy to use, generalized, name-based information store**. Use it to **track numbers and strings** for any logical (named) instance. Examples are keeping score, or tabulating cargo/weight (for both of which DML modules already exist and that utilize nameStats to work their magic)

#### 2.1.3.7 *cargoSuper*

This module can **manage inventories and weight** for logical (named) instance. It does not apply the weight to a unit, it simply provides a simple, abstract API to manage cargo items and their weight

#### 2.1.3.8 *cfxCargoManager*

A module that watches cargo and **creates cargo events** that whenever something noteworthy happen that relate to the **cargo it monitors**. Due to current limitations in DCS, this module synthesizes some of the events from changes in the objects it watches.

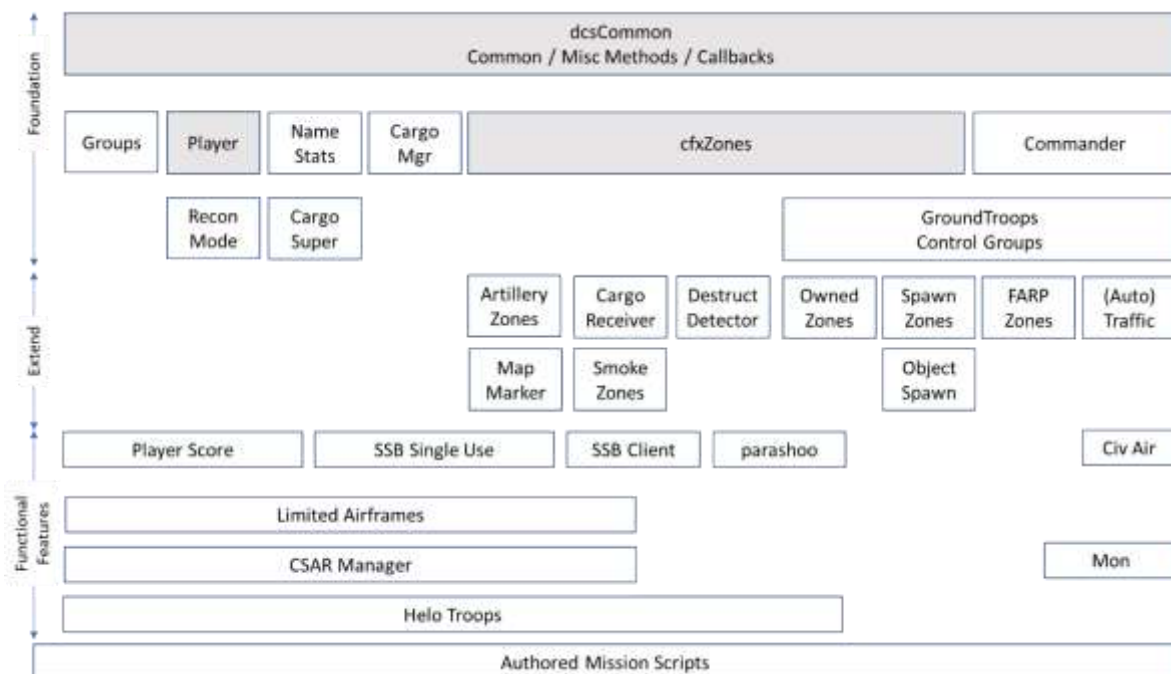
#### 2.1.3.9 *cfxGroups*

A module that reads the mission data at starts and provides a table for all initially defined groups. This module's main use is to **provide information about player mission 'slots'**.

## 2.2 Architecture (Lua Only)

Conceptually, imagine DML as a modular “upturned layer cake”: It starts with a Foundation layer of modules that provide common or miscellaneous functionality. Part of that layer is a collection of abstract modules (e.g., dcsCommon, cfxZones, cfxPlayer) that provide important services to all other modules and **provide integration with ME**. Together these modules are the ‘**Foundation Layer**’. The functionality in this layer is accessible to mission creators **only by means of Lua scripting**. Therefore, few people will ever use Foundation directly.

Below the foundation are modules that **Extend or Add Functionality**, by **combining their functions with Zones in ME** and provide ME-based means to access them. These modules allow a mission author to utilize new functionality **without writing a single line of code** (most of these modules provide hooks for optional script integration), simply by means of placing zones in ME and adding attributes. They also often serve as convenient building blocks for more advanced modules. Examples are Destruct Detectors, Artillery Zones and Spawn Zones.



Below the Extension layer are functional **Feature Modules** that provide ready-to-use functionality for your missions. They can be customized with Zones and Attributes, and can be interfaced with by scripts. Examples are CSAR Manager, Player Score and SSB Client.

The final (optional) layer are scripts that mission author create who choose to tap into any of the DML’s modules.



cf/x Dynamic Mission Library  
for DCS

## PART II: USING DML

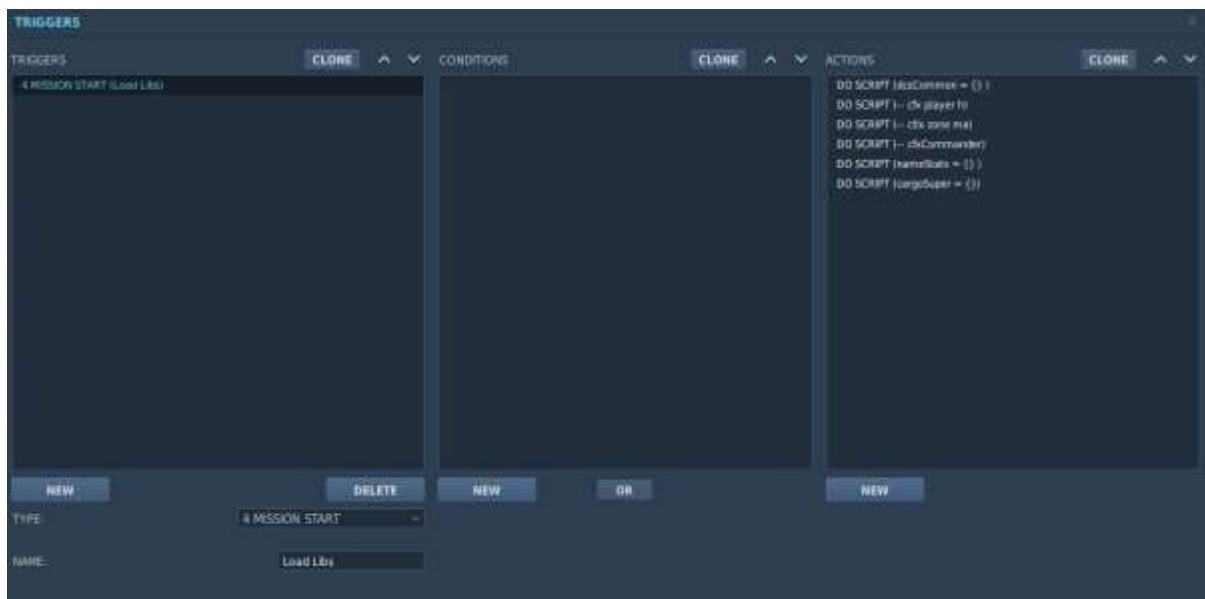
## 3 Using DML

This section describes how you bring individual DML modules into your missions, and how to use them. This chapter is divided into the following sections:

- *Importing Modules into Mission*  
Describes the concept of what modules are, and how you import modules into your mission.
- *Important Concepts*  
The library uses some central concepts that help facilitate creating missions. This section gives an overview of these concepts: Zones, Attributes, ME Flags, Configuration, Orders, Formations, Ownership
- *Using the Modules*  
Describes in principle the functionality of each module, their dependencies and properties

### 3.1 How to import Modules into a Mission

DML is organized in multiple “modules” which are nothing more than small text files. Each of these text files is named after the module they contain. To bring these modules into your mission, first create a MISSION START trigger. Then add DO SCRIPT actions to that trigger, one for each module that you want to include into the mission.

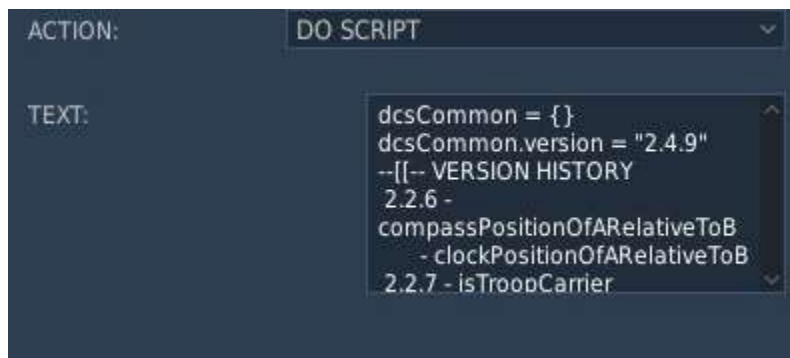


In above example, we have added six modules to the mission: dcsCommon, cfxPlayer, cfxZones, cfxCommander, nameStats and cargoSuper.

#### **Note:**

We'll discuss later what these modules actually do, here we look at how to bring them into a mission.

Next, open each module with a text editor (e.g., NotePad), and **copy/paste the entire text** from each module into their own DOSCRIPT text box.



The sequence of the actions determines the order in which the modules are loaded when the mission starts up. That is important to remember when there are dependencies between modules, i.e. when module A needs the functionality of module B, that module B must load before A does.

Most modules check their integrity when they load, and you will see warnings displayed when a module fails to load because it needs functionality to be loaded before it loads itself. This document always lists which module requires what functionality in their 'dependencies' section, so you can quickly look up which modules you need for your mission.

**Note:**

We use DOSCRIPT actions because of their simplicity. Other mission designers prefer a DOSCRIPTFILE action instead. Either will work, and for simplicity we'll stick with DOSCRIPT for the entirety of this document.

**Note II (advanced users)**

You may wonder why we don't simply pack all of DML into one big file that simple works for all missions with a unified START Trigger. That will surely work, and if you like that simplification, feel free to do so. The performance penalty is negligible. From an engineering perspective, however, it's highly displeasing to create a drab monolithic slab out of what is architecturally designed to be beautifully modular.

## 3.2 Important Concepts

Mission scripting can be challenging and extending the capabilities of the Mission Editor (ME) that is packaged with DCS World needs some careful advanced planning in order to make it as simple as possible. To do this, DML uses a number of easy-to-understand concepts that help integrate the modules with ME and simplify accomplishing certain mission goals like ordering troops, occupy areas etc.

### 3.2.1 Zones and Attributes

DML uses a central ME tool for integration: Trigger Zones. They can be placed anywhere on the map, are easy to modify (move, change, copy and paste), and they support a central feature that we use to pass information from ME to our modules: Attributes.

Attributes are named values (or “name/value pairs” in programmer parlance) that mission designers can add to, modify, and remove from Trigger Zones. An attribute (sometimes also called ‘property’) always has a name, and a value. Use ME to enter any text for both name and value.

#### Module Anchors

DML looks for attributes with certain names (e.g., “smoke”), and if it finds that attribute, automatically “anchors” the appropriate module to that zone (for example connects the smoke zone module to that zone). Read the “Using” section to find out which module looks for which attribute.

If you add an attribute and leave its value blank, *that* becomes its value (i.e. the attribute exists, it has the value <empty>).

The image on the right shows a Trigger Zone called “Red Two”. In the lower part a red box highlights the attributes that we added to this trigger zone. In general, you can add as many attributes to a zone as you like. **[Lua Only:** the cfxZones module gives designers easy access to a Zone’s attributes, can easily convert them, and collect all zones that have a certain property]

All modules use Trigger Zones with attributes to anchor modules, and to control a module’s functionality.

Usually, the name you give to a Zone (“Red Two”) itself is irrelevant; DML looks for specifically named attributes to anchor a module. You can therefore use the same zone to anchor multiple modules.

There are cases where a zone’s name is relevant to provide data: to configure a module, DML uses so-called Module Configuration Zones (see below) that can be omitted entirely (if you do not want to curtail the way a module works), and place anywhere on the map.



### 3.2.2 ME Flag integration into DML


Beside Trigger Zones, another central tool ME allows mission designers to control the flow of control: 'Flags'. Mission designers use Flags as a way to "persist" (remember) states.

Although Flags in ME are quite primitive, they can be used to great effect – as many existing missions show.

Many Zone Enhancements in DML can monitor a flag for change (which can trigger this module's action), or change a flag when they activate





#### Flag Query Attributes

Whenever you see a DML module that supports an attribute with the name "f?" that means that this module can query (and be triggered by) the ME Flag that you supply as value (here 100). If the value of that flag changes, the module's function (e.g., spawning) is triggered once. This can repeat as often as you like.

Name	Value	
f?	100	

#### Flag Set Attributes




When you see a DML module support an attribute that looks like a short mathematical formula like "f+1", it means that this module can change the value of the specified flag when it activates. On the right, the "f+1" attribute with a value of "110" means that when the module activates, it increases the value of ME flag 100 by one (hence "f+1"). Please refer to each module's "ME Attributes" section to see if the module supports reading or changing ME flags.

Name	Value	
f+1	110	
f=0	100	
f=1	200	
f-1	210	

#### Flag Bang! Attributes

When you see a DML module ask for an attribute that looks like it is shouting (r!), that means that the module changes the given flag depending on the situation. For example, the Owned Zones module supports a "b!" attribute.

This means that the ME Flag that is given as value to this attribute can both increase (when blue has conquered a zone) and decrease (when blue has lost a zone). The way that such a 'bang!' attribute changes is dependent on the module. Please refer to the module's documentation for details.

Name	Value	
b!	10	
r!	20	
n!	30	

### 3.2.3 Module Configuration Zones

Since Trigger Zones are so convenient, most modules also use them to provide the option to set configuration values, or to pass data for processing. In these cases, the zones also use attributes with values to pass data to a module, while the Trigger Zone's name itself is used to anchor the zone to the relevant module. Configuration zones are mission global – they control how a module works across the entire map.



In above example, we see three configuration zones: one each for cfxCommander ("CommanderConfig"), limitedAirframes ("limitedAirframesConfig"), and cfxGroundTroops ("groundTroopsConfig"). As mentioned, for configuration zones, their **name** is relevant: **it must match exactly** the name that is specified in the module's description.

Being able to control configurations with Trigger Zones makes it easy to curtail a module to your mission's requirements; all you need to do is add the relevant attribute to a correctly named zone, and your module is configured.


Unlike with most other zones, placement (location) of a configuration zones irrelevant, you can place them anywhere you like. A good place for them is somewhere out of the way where they can't confuse or get in the way (one of the corners of the map, for example). Some people also like to color-code config zones (we use yellow).

### 3.2.4 Orders

Ordering troops is a central ability in DCS. Some modules can produce (spawn) units, and are able to give, or pass on, orders. Orders are a DML concept that is not accessible from DCS ME, and using orders requires the presence of some DML modules.

Generally, orders are entered as attributes in the Zone enhancement that produces them (e.g. Spawners, Owned Zones), and then are handled while the modules pass group ownership between them. Some Order Attribute requires parameters. When needed, these are supplied as separate attributes. For example, the 'guard' orders require a parameter that tells the module at which range enemy troops are automatically engaged. For this, a separate 'range' attribute is added to the zone.

### 3.2.4.1 Available Orders

Orders	Description	Parameters
guard	Places the group in guard mode. It will actively look for enemies and, upon detecting them, will move towards and engage the enemies. After destroying all enemy units, the group goes back to guard mode. If given, range defines to what distance (in m) enemy ground units are detected.	range
attackOwnedZone	Automatically seek out the nearest enemy or neutral owned zone, and move to conquer it. If the zone is conquered while this group is still under way, it looks for the next closest owned zone. If there are no more owned zones, orders are switched to 'guard'	
attackZone	<p>Move to attack the zone referenced by name in the 'target' attribute. The name of a Zone is the same as you entered in the "Name" field for the Trigger Zone in ME at the very top.</p>  <p>So, to attack the zone defined above you would first enter "attackZone" as value for the "orders" attribute, and then enter "Red Two" as value for the "target" attribute. If the target zone can't be found, the group's orders are switched to 'guard'</p>	target
lase laze	<p>These units do not engage the enemy, but lase any enemy target that they detect up to a distance of the range parameter. Lase code is 1688 and currently can't be changed. Targets must have LOS, or they won't be lased.</p> <p>Just one of the units lases, the other units are back-ups if the lasing unit is killed.</p> <p>Units that have lasing orders interact with the jtacGUI script by passing target information and alerting players that lasing information is available.</p> <p>Order attribute can be named 'laze' or 'lase'</p>	range
training train dummy dummies	<p>All units are issued 'ROE HOLD' and will not engage any enemy. Once all units are destroyed, the entire group respawns after cooldown. This is useful for training missions where you want to set up self-replenishing enemy targets that don't fight back, for example for bombing schools.</p> <p>Order attribute can be named 'training' or 'train'</p> <p>DO NOT USE AUTOREMOVE with these orders, or you'll have lots of targets - quickly</p>	

#### 3.2.4.2 “wait-“ Prefix for orders

When units spawn, it's not always in the interest of the mission's design that they carry out their orders immediately. This is especially true for units that are intended to lase targets, or move to target zones only after they have been transported to their destination.

To temporarily stay an order, you can prepend the word “wait-“ to the orders (do not forget the hyphen). For example, when you want troops to lase targets after they have been transported, their orders for the spawner is “wait-lase”, instead of just “lase”. Once the troops have been transported, the ‘wait-‘ prefix is removed, and the orders are carried out. As long as the orders carry a ‘wait-‘ prefix, they are interpreted as ‘guard’ with default range.

### 3.2.5 Spawn Formations

When groups are spawned, they are assembled into a formation. You can tell the spawners what formation the group should assume. This is purely for the group's initial arrangement, should the group move, they will break that formation. A formation always assembles around a point and takes a second parameter that defines the area that the formation covers (size). For this the spawner usually takes the Zone's center and radius, but some spawners can work with polar coordinates and/or displacement (r, phi) to define where to assemble and at what size.

Formation	Description
line	A single file of units, left to right. <b>If there is only one unit, the center of the spawn zone is used as position.</b> Use this formation to place a single unit exactly where the spawner is located (most other formations start with the zone's periphery)
line_v	A single column of units
chevron	A chevron with the middle units most forward
scattered, random	Units are spread randomly across the zone
circle, circle_forward	Units are arrayed in a circle, all facing forward (same direction)
circle_in	Units are arrayed in a circle, all facing inwards towards center (like a huddle)
circle_out	Units are arrayed in a circle, all facing outwards (very good for SAM)
grid, square, rect	Units are packed into a grid with optimal (rectangle with smallest surface) fit for the number of units. Note that even if you specify ‘square’ are formation, it is not guaranteed that the units form a square
cols, 2deep	Units are arrayed in two columns
2wide	Units are arrayed in two lines





Above: 15 BTR-80 spawned in “grid” formation

### 3.2.6 Spawning: Type String and Type String Arrays

Spawning units requires some deeper DCS knowledge about units that can be difficult to come by; it is currently not covered in DCS’s documentation: Unit Type Designations, that DML and DCS variously refer to as ‘Type’, ‘Type Name’, ‘Type String’ or ‘Type Array’. It is a shot text (string) that uniquely identifies to the game engine which 3D model and weapons to use, and it can deviate significantly from how it is named in ME. For example, the internal ‘Type’ of for what is called “LUV HMMWV Jeep” in ME is “Hummer”. This means that to find the Type for the unit that you want to spawn, you must find an information source that can provide you with the correct type string. This is a possible source you may find helpful, as I used if for all type strings in this document:

<https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB>

Modules (and methods from the API) that spawn units request such a “Type String”, or “Type String Array” for the purpose of identifying what unit to spawn.

Since that the **correct value for this attribute is usually invisible** for ME users, it must be taken from (third party) documentation. As mentioned above, the “Type String” in DML corresponds to the “type” attribute in the spawn data table, and “typeName” attribute in the game’s object DB.

Since it’s desirable to spawn more than one unit per group, modules support a “Type String Array”. This is simply a string that contains multiple Type Strings (one for each unit), separated by comma “,”.

- For example, to spawn three Infantry Soldiers carrying the M4 and a single LUV HMMWV Jeep, use the Type String Array “Soldier M4, Soldier M4, Soldier M4, Hummer”.

Note that again, the type string value for “Soldier M4” was retrieved via external sources, just like “Hummer” before.

#### Important Note

You *can* insert **blanks to separate** the individual type strings visually (i.e. after the comma) - but be careful: do not insert blanks into the type string itself.

### 3.2.7 Ownership / Owned Zones

Owned Zones are DML zones that are in possession of a faction (note that this is a DML feature and not available in ME directly). Modules use this in zones that use ownership to control behaviors:

- Change in ownership can trigger a callback for some owned zones
- Spawners can stop spawning when controlled by the wrong faction
- Troops can automatically be ordered to seek out and attack the nearest owned zone
- FARP Zones automatically handle ownership change and spawn the correct resources

Owned zones are conquered by placing ground units inside the zone. If only one side has ground units inside an owned zone, ownership is transferred to that side, and stays with that side until only ground units from the other faction are inside the zone. Neutral units do not count, and a zone can be captured with a single unit even if there are neutral units inside.

Script Authors be advised that although there are many enhancements available to handle the ownership of a zone, the 'owner' attribute itself is provided (but not managed!) by the cfxZones foundation module, so you can access that attribute even if you do not include zone-based enhancements.

[Note: it's currently planned to separate the cfxOwnedZones into the ownership part, and the producing part that is currently integrated into cfxOwnedZones for a clearer separation of these concepts]

### 3.3 DML Mission Design Philosophy (Lua Only)

#### PLEASE BE ADVISED

If this is the first time you are reading this document, I **strongly recommend that you skip this rather technical section and move directly to →3.4 Using Zone Enhancements** to acquaint yourself with Zone Enhancements first and play around a few of the included demos. It will make understanding the “DML Way” much easier.

The Demos, Zone Enhancements, and Feature Enhancements should be your first steps exploring DML. You’ll eventually return here to learn the nitty gritty and how to coax the most out of DML. For now, just remember that this section exists.

DML uses a design philosophy common to many commercial game engines, and it may be helpful for mission designers to adopt a similar design approach in their own missions, as it dramatically speeds up mission design, modularity, and reduces test requirements.

Modern game engines organize around a couple of main principles:

- *Game States* that describe the main game “situations” (e.g. “assembling”, “ingress”, “attack”, “egress”)
- *Update Loop* that performs Game State-appropriate actions (e.g. generate random encounters during ingress) and may generate state-changing “Events”.
- *Event Handlers* that are invoked at very specific situations, called ‘Events’. Event handlers decide if one Game State should transition to another (all planes have taken off) and change the Game State accordingly.
- *Configuration Data* provided from ME (debugging, difficulty, etc)
- A *Start()* method to get everything in position and kick off updates

#### 3.3.1 Game States

You can always divide a mission into discrete phases, or “states”. You can usually easily define the conditions when a mission goes from one state to the other. One of the surprising discoveries is that – when done correctly – a mission usually has very few states, and that states follow a strict sequence; also, you will find that it’s quite easy to determine/define what makes a mission to transition from one state to another. Once you have determined the relevant states, it becomes easy to determine what to look for: the things that make a mission change its state.

#### 3.3.2 Update() Loop

This is a “main” loop that is invoked regularly – for example once every second. All it does is read the current *state* of your mission, and branch to the appropriate state handlers. Usually, it does nothing, as few states require constant hands-on management from the mission. The Update Loop thus usually consists of a few simple state checks followed by the occasional branch and invocation.

### 3.3.3 Event Handler

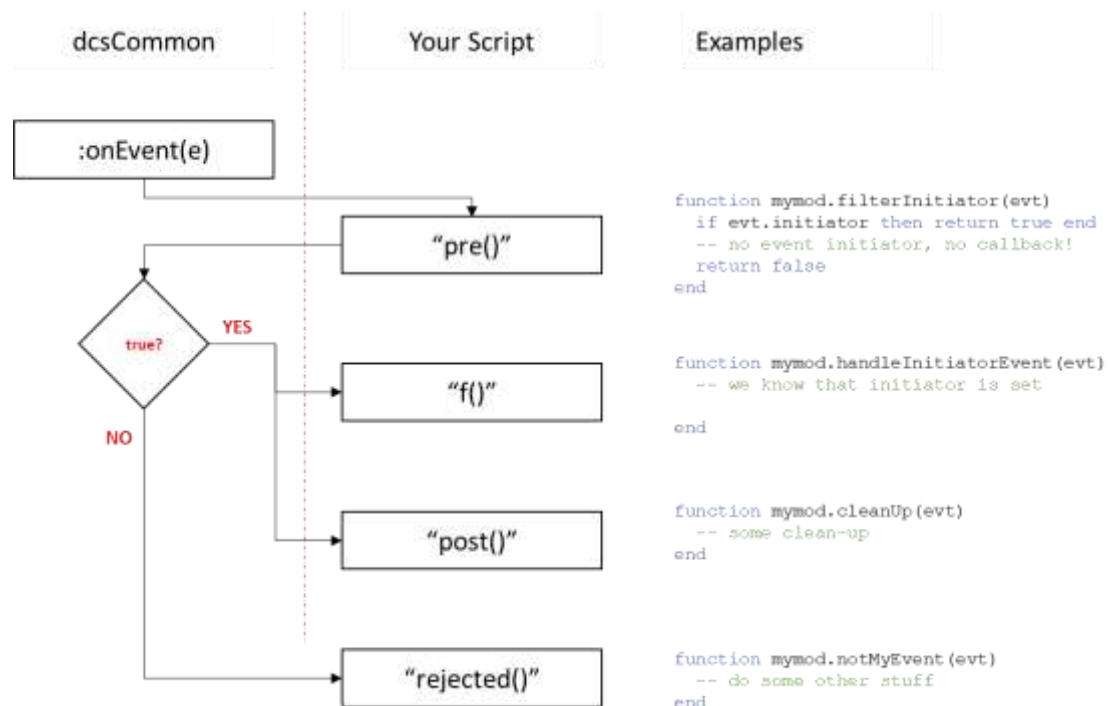
Events are pre-defined situations that can influence a mission's state, but not necessarily so (meaning: an event carries the possibility of changing the state, but does not necessarily always do so. For example, a 'take off' event by itself does not mean that a mission can progress to 'ingress' state, it may require that all planes of a package have taken off). Therefore, in addition to invoking a main update loop, most game engines also provide means for scripts to be told that certain events have occurred, and provide hooks for designers to place callbacks to be invoked in such cases: "event handlers".

As one of the central means to manage a game's flow of control, DCS of course provides its own event dispatcher: `world.addEventHandler()`. Script authors can use this to subscribe to world events and thus be 'in the loop' if something happens.

#### 3.3.3.1 *dcsCommon event dispatcher*

The 'bedrock' module `dcsCommon` (see later) provides a convenient, modular replacement for the one provided by DCS. Instead of implementing your own `onEvent()` methods on your own tables and calling `world.addEventHandler()`, you can use `dcsCommon`'s more advanced event handler that allows for the following features:

- *Use any method name for event processing*  
You can freely name your event responding method, it does not have to be `onEvent`
- *Optional pre-processor method*  
You can pass your own pre-processing method to analyze the event data. This pre-processor then decides (by returning true) if the event processor method is invoked.
- *Optional post-processor*  
You can pass your own function that is invoked only after event processing method was invoked. Handy for closing transactional brackets, clearing data etc.
- *Optional non-processor (rejected)*  
You can pass your own function that is to be invoked when the pre-processor decided that the event processor is not to be invoked.



### 3.3.4 ME-provided “Configuration Zones”

Some mission- or map-specific data is often the only information that changes between missions (e.g. the names of critical units, airfield names, difficulty settings, etc.). The same is true for important switches like turning on debugging, or disabling some feature. Data-only information should be moved to a GUI-based editor rather than source code. This is because the latter forces developers to change their source code for every map or scenario. Since DML provides easy access to zone properties, and because ME provides a nice, convenient GUI to add, modify and remove properties, we combine these to an ideal solution that we call ‘Configuration Zones’.

DML-internal modules use them heavily, and DML’s API provides mission designers with easy access as well. Here’s how a script reads all attributes (name/value) of a ME-created zone named “myData”;

```

local theZone = cfxZones.getZoneByName("myData")
local myData = cfxZones.getAllZoneProperties(theZone)

```

myData now contains a table with all name/value attributes that the mission designer added to that zone in ME. You can use this data just as if it was typed into a table somewhere in your code.

#### **WARNING:**

After collecting data from a zone into a table, instead of keeping this data in a separate table, you may be tempted to simply copy all attributes as configuration data into the main module. Let’s say you wanted to read code module skel’s configuration from myData. You may think of doing the following:

```
for attr, val in pairs (myData) do
    skel[attr] = val
end
```

That is a **very bad idea**, as (in a *best-case* scenario) you will inadvertently overwrite parts of your code. **Always individually validate each configuration attribute** in code, provide correct defaults, and **set module-configuration** variables **explicitly** instead:

```
skel.singleUse = cfxZones.getBoolFromZoneProperty(
    theZone, "singleUse", false)
```

### 3.3.5 The Start() Method

Everything gets rolling in a start method that

- Checks the integrity of the mission (is everything where it should be?)
- reads configuration data
- sets up the initial game state
- connects all event handlers
- and then gets the update ball rolling

You start a mission by invoking its start() method.

### 3.3.6 The “Main” Skeleton

Putting above together, we usually get a basic mission code “skeleton” that looks something like this (note that the event handler isn’t yet connected):

```
skel = {}
skel.state = "" -- sate of the mission
function skel.handleEvent(event)
    -- handle event
end

function skel.update()
    -- schedule next update invocation
    timer.scheduleFunction(skel.update, {}, timer.getTime() + 1)
    -- check states and act accordingly
    if skel.state == "hi there" then
        -- do something
    end
end

function skel.start()
    -- init variables
    skel.state = "hi there"
    -- read config zone data (not done here)
    -- hook up event handlers (see later)
    -- start update loop
    skel.update()
end

-- start the mission
skel.start()
```

Above script is already a fully functioning mission that has its update() method invoked once per second, and that mission designers can use to monitor a game’s state. The missing pieces are the event handler, and config loader - but hooking that up is similarly trivial.

DML modules provide callbacks for a (rather large) host of different events that you can choose from. Not all event handlers are structured identically, but the central premise is always the same: the event handlers determine if a state change is required, and the update loop handles the current state. If the main loop detects that a change in state is required (possibly by determining that a mission is complete), it usually synthesizes an event, and itself invokes the appropriate event handler.

When you look into the code of the various DML modules, you’ll find that they also implement this design: cfxZones uses it to update moving zones, ReconMode monitors aircrafts that way, etc. Most DML modules provide callbacks that you can tie into your own scripted event handlers to be notified when something interesting happens. As a result, your own mission code quite often resembles above skeleton, with a few additional lines thrown in

to handle a handful of specific events, and to determine when the obligatory 'Mission Complete' message is to be displayed.

**Note:**

See →DML Mission Template.miz – (Lua Only) for a practical, more expanded working example of the skeleton code (dmMain)





See →Landing Counter.miz for a working example of a mission with event handlers



### 3.4 Using Zone Enhancements

All Zone Enhancements provide key functionality that they attach to a Trigger Zone that mission designers place with ME. In order to find out which zone is intended for them, the modules look for key attributes in zones that tell them to attach their functionality to that zone.

For example, the Smoke Zones module looks through all Trigger Zones in a mission for an attribute called 'smoke'. If it finds such an attribute, it knows that this zone has data that tell it what to do (place smoke at the center of the zone, and use the color that is given as the value for the smoke attribute)

Name	Value	
artilleryTarget	One	
f?	100	
shellNum	17	
strength	700	

Using this simple mechanism allows DML several important features

- Use ME's GUI to place a module's functionality, including copy/paste to rapidly populate a map with zone enhancements
- Integration with ME Flags, as flags can be used to tell modules which flags to watch or modify if something interesting happens
- Use Trigger Zones to pass configuration/setup data – so a module's code does not have to be modified to curtail it for a mission.
- Stack multiple modules onto the same Trigger Zone – each module homes in on its own keyword; you can therefore use the same Zone for more than one module. If you take advantage of this ability, you must take care that if two modules use the same attribute name, it's value is compatible with both modules. A common attribute for many zones is the "verbose" attribute. If a Trigger Zone is shared by multiple modules that all support the "verbose" attribute, it is up to you to ensure that the value you choose is applicable to all.

Using zone enhancement is simple:

- Add the module and all dependent modules to your mission in a MISSION START trigger
- Add a Trigger Zone (or more)
- Add an attribute to that trigger zone, and name it as described in that module's "ME Attributes" section.
- (Optionally: add a config zone for that module to change a module's base behavior)

DML currently provides the following Zone Enhancements for your missions:

- **Smoke Zone**  
A zone that provides perpetual colored smoke at its center
- **Object Destruct Detector**  
Changes a ME-compatible flag when a map object is destroyed
- **Ground Unit Spawner**  
Dynamically (in-mission) spawns ground units

- **Object Spawner**  
Dynamically spawns objects and cargo
- **Cargo Receiver**  
Can receive dynamically spawned cargo and change ME Flags when cargo was successfully received
- **Artillery (Target) Zone**  
Rains destruction onto this zone when told to
- **Owned Zone**  
A zone that can change ownership (neutral/red/green) and therefore be conquered
- **FARP Zone**  
A FARP with an owned zone (see above) attached. Provides services (rearm, refuel, repair) after being conquered to the owning faction
- **Map Marker**  
Places text on the F10 in-game map
- **NDB**  
Places a non-directional (movable) beacon (NDB) that you can tune your ADF to. If the zone is set to move via a linked unit (e.g. a ship), the NDB's location updates regularly. NDB can be set to any frequency, AM and FM.

We'll describe each module in detail in the following sections.

### 3.4.1 All Zone Enhancements

Since all Zone enhancements use DML's foundation module "cfxZones" to anchor their functionality, all Zone Enhancements also inherit the foundation's core abilities, i.e. they all support the following attributes:

Name	Description
linkedUnit	Moves the zone's center with the unit who's name exactly matches the value of this attribute. That unit must exist at the beginning of the mission, or the zone will not be linked. Note that you <b>can</b> link units and zones after mission start by using the API
useOffset	Must be set to "yes" or "true" to have this effect, ignored otherwise. Only has an effect if the zone is linked. Keeps the offset between the unit and zone constant. Note that the zone's center remains the same in relation to the unit's center. If the unit turns, the offset does not change with the unit's heading.
(turnWithUnit)	(Currently not implemented, for later expansion of useOffset)
owner	The coalition that owns this zone. Used with many other zone enhancements. Some enhancements can even change this value  This attribute is added to all zones, even if not present. Default owner is neutral.

Some care must be taken when using inherited abilities, as they not always work as you may expect. The smoke zone, for example, when used with a linkedUnit attribute, results in smoke that jumps to a new location every 5 minutes, and so on.

### 3.4.2 cfxSmokeZones

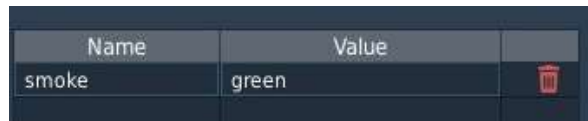
DCS provides a 'smoke' effect in various colors that unfortunately ends after a certain time. For missions it may be useful to be able to *permanently* mark a location with colored smoke.


cfxSmokeZones does exactly that: any zone that has the relevant attribute has an automatically "refreshing" smoke at the center with the color that is specified as the attribute's value. This module is primarily intended to be used via ME, but also provides an API

#### 3.4.2.1 Description

The zone receives a "never-ending" smoke effect of the specified color placed at the center. For this, the zones are placed in a list of managed zones that have their smoke effect refreshed regularly.

To add a smoke effect to the center of a trigger zone, simply add the 'smoke' attribute, and enter the desired color (green, blue, white, orange, or red) as value.



Name	Value	
smoke	green	

You can either add zones using ME (preferred), or using the API. You can only remove smoke zones from the managed list via API. Once you remove a zone, the smoke effect will not be renewed. This means that the effect usually does not disappear immediately, but when the smoke effect times out after the last refresh.

#### 3.4.2.2 Dependencies

Requires dcsCommon, cfxZones

#### 3.4.2.3 Module Configuration

None.

#### 3.4.2.4 ME Attributes

To add a permanent, colored smoke effect to a zone, add the following attribute in ME

Name	Description
smoke	Adds a permanent smoke affect to the center of the zone. Possible values for the smoke effect are: <ul style="list-style-type: none"><li>• "green" or "0"</li><li>• "red" or "1"</li></ul>

Name	Description
	<ul style="list-style-type: none"> <li>• “white” or “2”</li> <li>• “orange” or “3”</li> <li>• “blue” or “4”</li> </ul> <p><b>MANDATORY</b></p>

#### 3.4.2.5 API

In addition to ME you can use a simple API to start and stop a zone’s smoke effect. Unlike with ME, you *can* stop the effect with invocation of API methods.

##### 3.4.2.5.1 `addSmokeZoneWithColor(aZone, aColor)`

Adds `aZone` to the managed smoke zones. A colored smoke effect of `aColor` (a number 0 ... 4) is placed in the middle. The effect is maintained indefinitely, or until it is removed from the managed smoke pool via `removeSmokeZone()`. If `aColor` is omitted, the smoke effect’s color is green.

If invoked for a zone that is already managed, a second smoke of the new color is added, and eventually, after the old color times out, the new color remains.

##### 3.4.2.5.2 `removeSmokeZone(aZone)`

removes `aZone` from the list of managed smoke zones. Note that the smoke will not disappear instantaneously. Instead, the smoke effect will time out and disappear after some time.

#### 3.4.2.6 Using the module

Include the `cfxSmokeZones` source into a DOSCRIPT Action at the start of the mission

### 3.4.3 cfxObjectDestructDetector

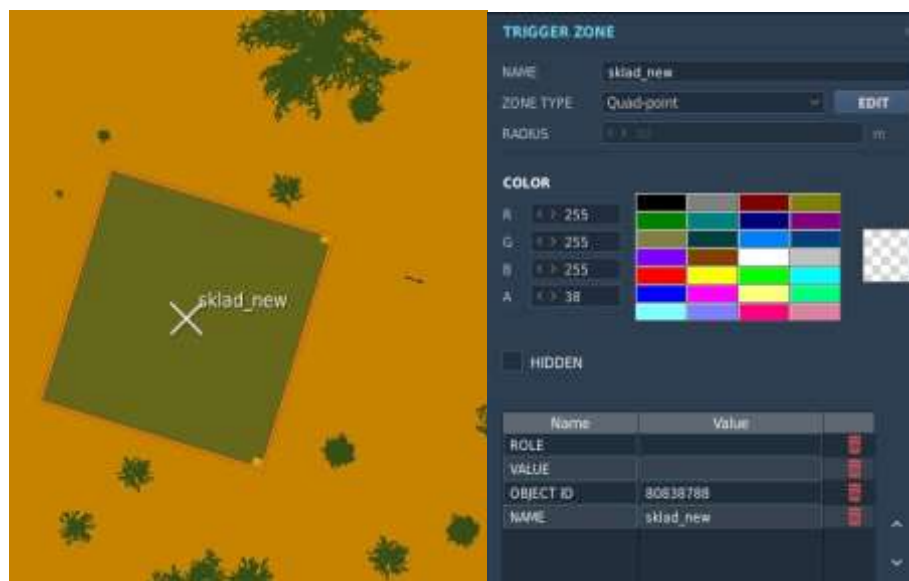
The destruct detector is a module that tightly integrates with ME, and additionally provides a scripting API. Unlike most other zone enhancements, this module provides direct means to manipulate ME-accessible flags.

#### 3.4.3.1 Description

A peculiarity of ME is its ability to create a zone that exactly fits around a map object, and automatically assigns some attributes. In order to create such a zone, simply right-click on a map object, and a small button 'assign as...' pops up.



When you click on the button, ME automatically creates a zone that exactly fits the shape of the object, and opens the zone editor, with several attributes added:



By default, ME adds the attributes Role, Value, Object ID and Name, with Object ID and name pre-filled: Object ID identifies the map object and is the ID scripts check to see if the object was destroyed.

When you include cfxObjectDestructDetector into your mission, it automatically seeks out all zones that have an "OBJECT ID" attribute, and starts watching them.

The beauty of destruct zones comes into play with new attributes mission designers can add to such a zone to automatically set, re-set or change flags without adding any code.

## ME INTEGRATION

You can add any of the following attributes to an object zone, and when the object referenced by OBJECT ID is destroyed, the module invokes the following:

Name	Value	Description
f=1	Number	Sets the flag <Number> (as accessed by DCS) to 1 (One) when the object is destroyed. If no <Number> is given, flag number 999 is set
f=0	Number	Sets the flag <Number> (as accessed by DCS) to 0 (Zero) when the object is destroyed. If no <Number> is given, flag number 999 is set
f+1	Number	Increases the current value of flag <Number> (as accessed by DCS) by 1 (One) when the object is destroyed. If no <Number> is given, flag number 999 is increased
f-1	Number	Decreases the current value of flag <Number> (as accessed by DCS) by 1 (One) when the object is destroyed. If no <Number> is given, flag number 999 is decreased

### NOTE

Since objectDestructDetector homes in on any zone with an OBJECT ID property, it automatically works with all objects that a mission designer marks with an *'assign as'* zone that automatically attached the OBJECT ID.

Thus, simply adding this module to a mission and then subscribing to the destruct callback (see below) is enough to interface your scripts to be notified of object destruct events for thusly marked objects.

### WARNING

The OBJECT ID that ME reveals with *'assign as'* is taken from an internal DB for that map. In the past, there have been instances where the object ID of *some* map building change between DCS releases. So, when you are using object destruct detectors and they suddenly stop working, check the object ID that the detectors track against a newly created "assign as" zone. If they differ, the map's internal Object DB has been updated and you must update the detector zone's object ID.

### Callbacks

For easy scripting integration, destruct detector provides events callbacks whenever an object described in OBJECT ID is destroyed. The callback must match the following profile:

```
theCallBack(zone, ObjectID, name)
```

with `zone` being a link to the `cfxZone` (the DML wrapper for the DCS Zone!), `ObjectID` the DCS object ID of the object that was destroyed, and `name` the value of the name attribute.

### NOTE:

Object Destruct Detector only generates events for object ID's that are defined in ME via the *'assign as'* function. The module will not detect destruction of objects other than the ones it is told to watch.

#### 3.4.3.2 Dependencies

This module requires dcsCommon and cfxZones to be loaded

#### 3.4.3.3 Module Configuration

`cfxObjectDestructDetector.verbose` – set to true to see a message each time a watched object is destroyed.

#### 3.4.3.4 ME Attributes

As described in “ME Integration”, destruct detector supports multiple attributes that tell it what to do (besides invoking callbacks) when a watched object is destroyed.:

Name	Description
OBJECT ID	THIS ATTRIBUTE IS FILLED BY ME AND MUST NOT BE CHANGED <b>MANDATORY</b>
NAME	THIS ATTRIBUTE IS FILLED BY ME AND MUST NOT BE CHANGED <b>MANDATORY</b>
f=1	Sets the flag specified in Value to 1 when object is destroyed
f=0	Sets the flag specified in Value to 0 when object is destroyed
f+1	Increases the value of the flag specified in Value by 1 when object is destroyed
f-1	Decreases the value of the flag specified in Value by 1 when object is destroyed

#### 3.4.3.5 API

You can use the API to intercept all destruction events for any object marked with an OBJECT ID in a zone

##### 3.4.3.5.1 `addCallback(theCallback)`

Adds theCallback to destruct detectors list of methods to invoke when a watched object is destroyed.

#### 3.4.3.6 Using the module

Include the `cfxObjectDestructDetector` source into a DOSCRIPT Action at the start of the mission

### 3.4.4 cfxSpawnZones

#### 3.4.4.1 Description

cfxSpawnZones is a Zone Extension that allows you to add Group Spawners (i.e. points on the map where new ground units are created in-mission) to your mission. By spawn zone your mission can spawn groups dynamically (i.e., at runtime).

A spawner can spawn once, a set number of times, on demand, or indefinitely. A spawn cycle each time spawns a group composed of the vehicles/infantry that are defined by the 'types' attribute. You can find a good reference of the type strings for each individual unit here: <https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB>

#### Spawn Cycle

General spawning logic is that, unless paused, a spawner produces a group and places it inside the spawn zone radius according to the 'formation' attribute. When the group is removed from the spawner's control (be it automatically, by being destroyed, being picked up by a transport (including aircraft via cfxHeloTroops), or ordered to move out), the spawner undergoes a 'cooldown' cycle (waits for an amount of time) after which it produces the next group. Note that on mission start, an active (un-paused) spawner will spawn immediately.

cfxSpawnZones supports different spawning "behaviors" controlled by attributes, and has built-in capabilities to interact with other modules, e.g. GroundTroops (orders) and HeloTroops (airlift).

Spawning can also be controlled (paused) by the faction status of an associated controlling zone (via the masterOwner attribute: when the owner of that zone is a different faction, spawning is stopped). This is useful to control spawn availability on FARPS and airfields, or producing troops after a zone is conquered. Note that the spawn zone does **not** have to be within their associated masterOwner's zone radius, it can be on an entirely different place of the map.

#### ME Integration (forced spawns)

Spawners can be instructed to spawn immediately, at which point they ignore all of the rules programmed into them by attributes, and create a fresh 'batch' of troops immediately. SpawnZones can be told to watch an **ME flag**, and every time that flag changes, the Spawner spawns new units without checking max spawns, updating spawn count, or respecting a cooldown, not even ownership. In order to use an ME flag to trigger a spawn, all you need to do is add an attribute to the spawner:

Name	Value	Description
f?	Number	Watches the flag <Number> (as accessed by DCS) for a change. <b>Each time the flag value changes, a new group is spawned</b>

This simple mechanism allows mission designers to, for example, spawn troops whenever a player unit enters a zone (for very nasty surprises)

Alternatively (Lua only), scripts can use the API's method `spawnWithSpawner()` to directly trigger a spawn, also bypassing all checks.

After a forced spawn, SpawnZones resets the cooldown and invokes all subscribed callbacks.

#### Spawn Locations

Unlike ME, a SpawnZone does not care where it spawns the units. This means that you



must be careful not to place a spawner on surfaces that are too steep, or cause units to spawn in water (unless that is your objective). This can, however, be used for a nice exploit: you can spawn troops on off-shore platforms if you are careful enough with your positioning and the platform does not move. In some off-shore objects (like oil platforms), the units will fall through (the object has no hit box, as sadly some scenery objects don't have), in others, they stay in position and respond normally to enemy action



### Callbacks (Lua Only)

`cfxSpawnZone` supports callbacks that are invoked when a new group is spawned. To register a callback, invoke

```
function cfxSpawnZones.addCallback(theCallback)
```

The callback has the following signature

```
theCB(reason, theGroup, theSpawner)
```

with `reason` being string describing why the callback was invoked, `theGroup` being the newly spawned DCS group, and `theSpawner` the `cfxSpawnZone` table that was used to spawn.

Currently, the following reasons are defined:

- "spawned"  
SpawnZone theSpawner has just spawned theGroup. If the `cfxGroundTroops` module is installed, theGroup is passed to GroundTroops for management.

You can force a spawn by directly invoking

```
function cfxSpawnZones.spawnWithSpawner(aSpawner)
```

which will override any restrictions, and spawn the specified types immediately

#### 3.4.4.2 Dependencies

**Required:** `dcsCommon`, `cfxZones`

**Optional:** `cfxGroundTroops`, `cfxHeloTroops`

### 3.4.4.3 Module Configuration

This module does not need to be configured

### 3.4.4.4 ME Attributes

Name	Description
spawner	Marks this ME Zone as a spawn zone. <b>Value of this attribute is ignored</b> , use it to describe what it spawns to make mission editor easier for you <b>MANDATORY</b>
f?	Flag (ME-compatible) to observe. Each time the value of that flag changes, a new spawn is forced, ignoring all other settings like maxSpawn, cooldown, paused, etc. Defaults to no flag to observe
types	Type string array for the ground units that are spawned. Example "Roland ADS, Roland Radar, Roland ADS" or "Soldier M4" – <b>WARNING:</b> Blanks are part of the type, and blanks before and after the last character are automatically stripped. For a full reference of objects and their types, see here <a href="https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB">https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB</a> and use whatever is given as value for the "typeName" attribute, e.g. "Soldier M249" for the "INF Soldier M249.lua"
country	The country (a number) the units that spawn belong to, e.g. "22" for Switzerland ( <b>Warning:</b> unlike many other zone extensions, we use a County, not a Coalition here. The coalition is determined by which Faction the country belongs to as is defined when you create the mission, or by using the faction editor. Common Countries are Russia = 0, Ukraine = 1, USA = 2, UN Peace Keepers = 82 You can find a reference of all country codes here: <a href="https://wiki.hoggitworld.com/view/DCS_enum_country">https://wiki.hoggitworld.com/view/DCS_enum_country</a> ).
masterOwner	A string that references another ME Zone by name. It must match that Zone's name exactly, and that zone must have an owner (e.g. defined as an cfxOwnedZone or FARPZone). A spawner only spawns automatically when the masterOwner's owning faction is the same as the spawner's country affiliation. On the map, the spawner does not have to be inside the masterOwner's zone, it can be hundreds of miles away. You can use this to start spawning reinforcements in a completely unrelated part of the map when units conquer the masterOwner zone. If no masterOwner is specified, the Spawner spawns as directed and disregards any surrounding zones that happen to be owned Optional, defaults to empty
baseName	A name (e.g. "Hill Marines") that is used to create units and groups from during unit spawning. <b>If provided, baseName MUST BE UNIQUE. If you do not assign a base name, a unique one will be generated for you.</b> If two spawners have the same baseName, one of them will not spawn, so if for some reason a spawner does not spawn, make it a habit to check this first.
cooldown	Time interval (in seconds) from when a new group can be produced (removed from the spawner) to the moment it is produced. Defaults to 60

Name	Description
autoRemove	Usually, a spawner retains ownership of a group that is produced, and will re-start the spawning cycle only after it was removed. If you add the autoRemove attribute with a “yes” or “true” value, the Spawner will automatically re-start the spawning cycle (cooldown, produce) as soon as the new group has spawned. You can use this to automatically give orders and have units move out after they have spawned (similar to how OwnedZones spawn attackers). Be advised that you can create a lot of vehicles on your map in a very short time, so be careful when using autoRemove. Defaults to ‘false’
heading	The direction the spawned group is oriented to, from the center of the spawn zone. Defaults to 0
formation	Formation of the spawned group. See dcsCommon for supported group formations. Defaults to ‘circle_out’.
paused	When paused, a spawner only spawns when other scripts tell it to (e.g. your own scripts, cfxHeloTroops, triggers). Defaults to “no”
orders	This is an optional interface to other troop-governing modules, e.g. cfxGroundTroops. Default is “guard”, and spawners support in addition to those that cfxGroundTroops support (see → Orders)
range	An attribute used to pass a range value to orders (e.g. JTAC laze range, detection/engage range)
target	An attribute used to pass a target zone when used in conjunction with the ‘attackZone’ orders
maxSpawns	The maximum number of times that this spawner spawns groups. Set it to a positive number (e.g. 3) to spawn that many times. Set it to a negative number for an unlimited number of spawns (default is -1). Set it to zero (0) and the spawner will never spawn.
requestable	Interfaces with other scripts, if you set this value to true, troops will only spawn on request via <code>cfxSpawnZones.spawnWithSpawner()</code> . See the API section on how to get a list of eligible spawners. Automatically interfaces with HeloTroops and other enhancements

#### 3.4.4.5 API

In addition to ME Integration, SpawnZones support API to cause a SpawnZone to spawn, get SpawnStatus and receive callbacks when a SpawnZone is spawning.

##### 3.4.4.5.1 function `cfxSpawnZones.addCallback(theCallback)`

Adds theCallback to the list of callbacks to invoke when a spawner spawns. theCallback must match the following signature `theCB(reason, theGroup, theSpawner)`

##### 3.4.4.5.2 function `cfxSpawnZones.spawnWithSpawner(aSpawner)`

Causes aSpawner to ignore all restrictions (including cool-down and maxSpawn and spawn a group as specified in the types attribute with orders. aSpawner can be a string, in which case it must be the name of the zone as defined in ME.

Spawn callback is invoked, cooldown is reset, but the number of spawns is not updated.

#### 3.4.4.5.3 function `cfxSpawnZones.getSpawnerForZone(aZone)`

Returns the SpawnZone for aZone (a `cfxZone`), or nil if aZone is not a SpawnZone.

#### 3.4.4.5.4 function `cfxSpawnZones.getRequestableSpawnersInRange(aPoint, aRange, aSide)`

Given aPoint and aRange, this method returns a table of all SpawnZones that have requestable set true, and that are currently owned by aSide

#### 3.4.4.5.5 function `cfxSpawnZones.verifySpawnOwnership(spawner)`

Tests if spawner's ownership agrees with that of the master zone. If no master zone is defined, it returns *true*. This method only returns false if a master zone is defined, **and** that master zone's ownership disagrees with the coalition defined for spawner.

#### 3.4.4.6 *Using the module*

Include the `cfxSpawnZones` source into a DOSCRIPT action at the start of the mission

### 3.4.5 cfxObjectSpawnZone

#### 3.4.5.1 Description

ObjectSpawnZones are similar to cfxSpawnZones in that they are used to dynamically spawn objects into a running DCS mission (i.e., they can create objects that did not exist in ME when the mission started). Like their name indicates, ObjectSpawnZones are used to spawn “inanimate” objects into the game. These usually are cargo objects, but they can be used to spawn other static objects into the game. In DCS terms, ‘static objects’ are inanimate: they do not cause world events (like “dead”), can’t be controlled by AI, and will therefore not move by themselves, nor fight or otherwise react to the presence of enemy units - even if they look exactly like (non-static) units.

Since to DCS they are inanimate, they *can* be linked to other units (ships) and picked up as cargo by helicopters. ObjectSpawnZones has provisions to allow both: they can be linked to ships so that the objects that they spawn can be placed on the deck of ships (and then move with the ship), and the spawned objects can be declared to be cargo objects so helicopters can pick them up.

There are some small differences to Unit/Group SpawnZones, so make sure that you consult and understand the various ME Attributes.

#### Spawn Cycle

After objects are spawned, the ObjectSpawnZone keeps a look on the spawned objects. Once all of them have disappeared from the game (by deleting/destroying them), a new spawn cycle begins with a cooldown first, and then spawning all objects as described. Note that picking up cargo objects does not remove them from the game, so the spawner will not re-spawn simply because a cargo object was picked up. CargoReceivers (see below) have the ability to auto-delete cargo on deliver so this can then trigger the spawner’s re-spawn cycle.

When autoRemove is set to true the spawner immediately undergo a new spawn cycle after spawning.

#### Spawning “Formation”

Objects in object spawner always spawned objects as follows

- In the zone’s center if the `count` attribute is omitted or set to one (1)
- An evenly spaced circle on the perimeter of the zone if count is set a value greater than one.

#### ME Integration (forced spawns)

Spawners can be instructed to spawn immediately, at which point they ignore all of the rules programmed into them by other attributes, and spawn objects immediately.

ObjectSpawnZones can be told to watch an **ME flag** for change, and every time that flag changes, the spawner spawns anew without checking max spawns, cooldown. In order to use an ME flag to trigger a spawn, all you need to do is add an attribute to the object spawner:

Name	Value	Description
f?	Number	Watches the flag <Number> (as accessed by DCS) for a change. <b>Each time the flag value changes, a new set of objects is spawned</b>

This allows mission designers to spawn objects whenever a player unit enters a zone (e.g. cargo containers for helicopters)

Alternatively (Lua only), scripts can use the API's method `spawnWithSpawner()` to directly trigger a spawn, also bypassing all checks.

After a forced spawn, `SpawnZones` resets the cooldown and invokes all subscribed callbacks. Unlike Troop Spawns, a forced spawn does count against `maxSpawns`, but a limit overrun is ignored.

### Callbacks and scripted spawns (Lua Only)

`ObjectSpawnZones` support callbacks that are invoked when a new group is spawned. To register a callback, invoke

```
cfxObjectSpawnZones.addCallback(theCallback)
```

The callback has the following signature

```
theCB(reason, theSpawns, theSpawner)
```

with `reason` being string describing why the callback was invoked, `theSpawns` being a table of the newly spawned objects, and `theSpawner` the `ObjectSpawnZone` table that was used to spawn.

Currently, the following reasons are defined:

- "spawned"  
    `ObjectSpawnZone` theSpawner has just spawned theSpawns. If the module `cfxCargoManager` is installed and `isCargo` is set to true, the spawned objects are passed to `CargoManager` for management.

You can force a spawn by directly invoking

```
cfxObjectSpawnZones.spawnWithSpawner(aSpawner)
```

which will ignore any restrictions, and spawn immediately. A forced spawn does not count against the `maxSpawn` limit.

### Spawning Cargo

Objects can be spawned as cargo that can be then picked up by other units (e.g. helicopters). If you set the `isCargo` zone attribute to true, the object is spawned as a cargo object in DCS and responds to normal cargo commands. Make sure to also set the `weight` attribute in this case to control the cargo's weight.

Note that if you have installed the `cfxCargoManager` module in the mission, all **cargo is also automatically registered with the cargo manager** to generate cargo events that your script can subscribe to. In order to not register a spawned cargo object with cargo manager, set the `managed` attribute to false.

### Linking spawned Objects to Units (autoLink)

DML supports linked zones: zones that move with objects. Since a common behavior with spawned objects is that an object that is spawned from an Object Zone that is linked to a unit should also move with that unit (e.g. a cargo spawner placed on a ship), ObjectSpawner's default behavior for objects spawned with an ObjectSpawnZone that is linked is to also link the spawned objects to the unit that the object is linked to.

In order for moving (linked) object spawners to 'drop' their spawned objects to the ground (instead of onto the linked objects), add an autoLink attribute and set it to false. If no autoLink attribute is present, any object created from an object spawner that is linked to a unit is automatically also linked to that same unit.

#### 3.4.5.2 Dependencies

##### Required

ObjectSpawnZones requires dcsCommon, cfxZones

##### Optional:

cfxCargoManager (for managing cargo events).

#### 3.4.5.3 Module Configuration

ObjectSpawnZones does not require any configuration

#### 3.4.5.4 ME Attributes

Name	Description
objectSpawner	Marks this ME Zone as a spawn zone. <b>Value of this attribute is ignored</b> , use it to describe what it spawns to make mission editor easier for you <b>MANDATORY</b>
f?	An ME-compatible flag (e.g. 100) that this object spawnere monitors for change. Whenever the value of the monitored flag changes, a new set of objects is spawned immediately, ignoring all maxSpawn and cooldown rules.
types	Type string array for the STATIC OBJECTS that are spawned. Example "White_Tyre, Red_Flag". These objects may look like units (if you use the type string for a ground unit or aircraft), but they are static.  <b>WARNING:</b> Blanks are part of the type, and blanks before and after the last character are automatically stripped. All static objects given here are stacked on top of each other, and count as one instance (the example creates a tire with a red flag in the middle) <b>MANDATORY</b>
count	The number of times that the combined object in types is to be repeated. If count equals one (or is omitted), the objects defined in types are assembled in the center of the zone. Otherwise, the obejcts are distributed over the zone's circumference count times. Defaults to one
country	The country for which the static objects are spawned. Examples: 0 = Russia, 1 = Ukraine, 2 = USA etc. Defaults to 2 (USA)

Name	Description
baseName	Used to create the names that uniquely identify the objects that are spawned to DCS. <b>If provided, MUST BE UNIQUE</b> for each spawner. If you do not provide a baseName, a unique name is generated for you.
cooldown	Number of seconds after the last spawn was removed before new objects are spawned. Default is 60 seconds
autoRemove	Wait for the spawned objects to be removed or destroyed, immediately start cooldown, then re-spawn according to rules. Default is false
autoLink	Only used when the spawner is linked to a unit: should the spawned objects move with the unit that the zone is linked to (usually ships, but can also be other objects). Defaults to true. Set to false if the spawner should 'drop' the objects to the ground.
heading	Orientation of the objects when they are spawned. Default is 0 (North)
weight	Used with cargo objects: the weight of this object in kg. Defaults to zero.
isCargo	Are these objects to be picked up by helicopters? Defaults to false.
managed	Used only if the objects spawned are cargo. If true, cargo objects are automatically registered with cfxCargoManager when they are spawned and cfxCargoManager is loaded). Defaults to true
maxSpawns	Number of times that the spawner spawns the objects. Defaults to 1 (one)
paused	A paused spawner will not spawn automatically (but can be forced to spawn via API). Set to true to pause spawning. Defaults to false.
requestable	This spawner should only spawn on request (i.e. via API or from other zones). Forces paused to true. Default value is false

#### 3.4.5.5 API

In addition to configuring spawn zones with ME, mission designers can use the API for even finer control

##### 3.4.5.5.1 addCallback(theCallback)

Adds theCallback to the list of callbacks that are invoked whenever an object spawn event occurs. theCallback has to match the following profile: `theCB(reason, theSpawns, theSpawner)`

##### 3.4.5.5.2 getSpawnerForZone(aZone)

Returns the object spawner that is attached to cfxZone aZone. If no object spawner is attached to that zone, nil is returned.

##### 3.4.5.5.3 getSpawnerForZoneNamed(aName)

Returns the object spawner that is attached to the zone with the name aName. If no object spawner is attached to that zone, or if no zone with that name exists, nil is returned.

##### 3.4.5.5.4 getRequestableSpawnersInRange(aPoint, aRange, aSide)

Returns a table of all the object spawners that are inside aRange of aPoint, and that spawns object for aSide.



#### 3.4.5.5.5 `spawnWithSpawner(aSpawner)`

Forces `aSpawner` to spawn immediately, ignoring all current restrictions. If a cooldown timer is running, cooldown is reset. This spawn cycle is not counted against `maxSpawns`.

#### 3.4.5.5.6 `despawnRemaining(spawner)`

Removes all current objects from the list of spawns tracked by the spawner, leading to a new spawn cycle (cooldown, spawn) when the correct conditions are met.

#### 3.4.5.6 *Using the module*

Include the `cfxSpawnZones` source into a DOSCRIPT action at the start of the mission

Remember to also include `cfxCargoManager` if you want it to automatically managed cargo events

### 3.4.6 cfxCargoReceiverZone

#### 3.4.6.1 Description

This module solves a limitation of ME: unlike ME, it *can* generate events and set flags when players unhook cargo in such a zone. CargoReceiverZones provides strong integration for ME (via ME flag manipulation when something was delivered). Additionally, the receiver zones can provide automatic directions for the helicopter pilot during the final delivery phase.

CargoDeliveryZones work closely together with ObjectSpawnZones (who usually spawn the cargo objects) and the cfxCargoManager module that tracks the cargo objects and provides the required cargo events.

#### ME Flag Manipulation

Similar to the object destruct detector module, cargo receiver zones can manipulate standard ME flags (set, clear, increase and decrease), allowing mission designers not only to trigger on a delivery, but also use a single flag to count deliveries. This is controlled by adding attributes to the zone:

Name	Value	Description
f=1	Number	Sets the flag <Number> (as accessed by DCS) to 1 (One) when cargo is delivered. If no <Number> is given, flag number 999 is set
f=0	Number	Sets the flag <Number> (as accessed by DCS) to 0 (Zero) when cargo is delivered. If no <Number> is given, flag number 999 is set
f+1	Number	Increases the current value of flag <Number> (as accessed by DCS) by 1 (One) when cargo is delivered. If no <Number> is given, flag number 999 is increased
f-1	Number	Decreases the current value of flag <Number> (as accessed by DCS) by 1 (One) when cargo is delivered. If no <Number> is given, flag number 999 is decreased

#### Callbacks (Lua Only)

In addition to attribute-based flag manipulation, the module supports callbacks whenever cargo was delivered into a cargo receiver zone. The callback must match the following profile:

```
function cargoReceivedCB(event, obj, name, zone)
```

with `event` being a string describing the event, `obj` being the cargo object itself, `name` being that object's name (which can persist beyond the existence of the cargo object itself) and `zone` being the cargo zone as defined in ME

Currently, the following events are defined:

- “deliver”  
The cargo object `obj` was delivered into the cargo receiver zone referenced by `zone`

#### 3.4.6.2 Dependencies

CargoDeliveryZones can only track cargo that is registered with cfxCargoManager

It therefore requires that the following modules have loaded:

dcsCommon, cfxZones, cfxPlayer, cfxCargoManager

You usually also want cfxObjectSpawnZones to load because they can create cargo objects for you

#### 3.4.6.3 Module Configuration

(none)

#### 3.4.6.4 ME Attributes

Name	Description
cargoReceiver	Marks this zone as a cargo receiver zone. Value is ignored <b>MANDATORY</b>
autoRemove	Delete any object immediately after it was successfully delivered. This is helpful for most ObjectSpawnZones set-ups to trigger their spawn cycle
silent	Set to true to turn off this zone's directions. Defaults to false (zone will talk to pilots)
f=1	Sets the flag specified in Value to 1 when cargo is delivered
f=0	Sets the flag specified in Value to 0 when cargo is delivered
f+1	Increases the value of the flag specified in Value by 1 when cargo is delivered
f-1	Decreases the value of the flag specified in Value by 1 when cargo is delivered

#### 3.4.6.5 API

In addition to configuring cargo receiver zones with ME, mission designers can use the API for even finer control

##### 3.4.6.5.1 `addCallback(cb)`

Adds the callback `cb` to this module's list of active callbacks. Must match the following profile:

```
cargoReceivedCB(event, obj, name, zone)
```

#### 3.4.6.6 Using the module

Include the `cfxCargoReceiverZone` source into a DOSCRIPT action at the start of the mission

Place cargo receiver zone as your mission requires

### 3.4.7 cfxArtilleryZones

#### 3.4.7.1 Description

artilleryZones (better: artillery *target* zones, as they designate where the artillery shells will land) are a simple extension for mission builders that can be used to simulate artillery bombardment on a point on the map (without having to place artillery units), as well as marking an artillery zone visually (via smoke) and on the F10 map. In conjunction with the artilleryUI module, mission designers can easily implement forward observation (FO) methods for helicopters with support for spot range and LOS.

Artillery zones provide enough firepower (controlled with the shellStrength attribute) to destroy any object, so they are a good (and spectacular) choice to use when you need to destroy map objects (bridges, buildings). This can be further utilized with object destruct detectors that can tell you when a map object was destroyed (and stop bombardment), or trigger further bombardment to make sure an object gets destroyed).

Artillery zones can use standard mission flags to trigger a bombardment, so a mission designer can rig artillery target zone very precisely and then simply change a flag to start bombardment.

Finally, artillery zone has a comprehensive API for those who want to interface to artillery zones via scrip.

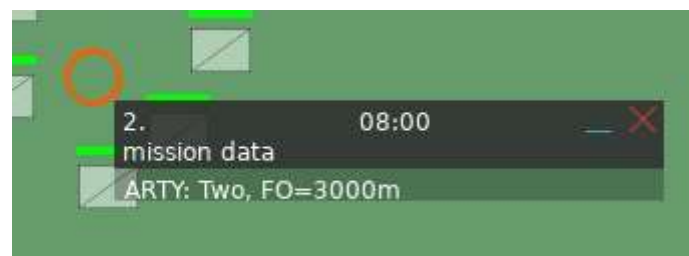
#### ME Flag Integration

You can trigger firing into the artillery zone with normal ME flags. You tell the artillery zones which flag to watch, and every time that flag changes value, a fire cycle is triggered. Note that flag-triggered firings ignore any cooldown attribute; when the flag changes, the artillery fires.

Name	Value	Description
f?	Number	Watches the flag <Number> (as accessed by DCS) for a change. <b>Each time the flag value changes, a new fire cycle is initiated</b>

#### Map Integration

Artillery target zones are marked on the F10 map for the coalition that this artillery zone belongs to (an optional attribute, see below). If no coalition is specified for the artillery zone, it won't be visible on red's nor blue's map.



Automatically marking an artillery zone can be suppressed with an attribute (see below)

#### Callbacks (Lua Only)

artilleryZones support callbacks. Your callbacks must match the following profile:

```
function artyCallback(reason, zone, data)
```

with

- `reason` describing why the callback is invoked
- `zone` being the artillery zone that is responsible for the event
- `data` being a table containing additional information for the event

Artillery zones invoke subscribes callbacks for the following reasons:

- `'fire'`  
The artillery is firing according to parameters. The data table is empty.
- `'impact'`  
A (simulated) projectile from a fire event has impacted. The `data` table holds the additional information: `data.point` is the impact point of the shell, `data.strength` is the power of the explosion  
Note that for each fire event there can be multiple impact events for the same zone: that zone's `shellNum` invocations to be precise.

You can subscribe to artilleryZone callbacks by invoking

```
cfxArtilleryZones.addCallback(theCallback)
```

#### 3.4.7.2 Dependencies

`cfxArtilleryZones` requires `dcsCommon` and `cfxZones`

#### 3.4.7.3 Module Configuration

None.

#### 3.4.7.4 ME Attributes

`cfxArtilleryZones` make heavy use of attributes. Make sure to understand the defaults; usually you'd only need to change some of the attributes

Name	Description
<b>artilleryTarget</b>	Marks this zone as an artillery zone. Value is ignored <b>MANDATORY</b>
coalition	Used with Artillery UI – the coalition that can give a fire command (the explosions are completely coalition agnostic – they kill anyone). When the artillery zone is marked on the map, only this side will see it. Defaults to 0. Supports “red” and “blue” as values
spotRange	Used with Artillery UI – the maximum range at which an FO can give a fire command. Measured from center of zone. Defaults to 3000 meters
shellStrength	Average power of <b>each</b> exploding shell. Defaults to 500. 3000 is enough to level big buildings, so be conservative.
shellNum	Number of shells (salvo) per fire cycle. Defaults to 17 shells per cycle

Name	Description
transitionTime	The time (in seconds) the shells take on average to reach the target zone. Note that not all shells arrive at once, but are usually spread over a couple of seconds. Defaults to 20
addMark	Add the artillery target zone to the F10 map of coalition (see above). Defaults to <b>true</b> .
shellVariance	Difference in shell's explosion power, in percent. Defaults to 0.2 (20%)
triggerFlag f?	The ME flag to watch to trigger a fire cycle. Whenever the value of that flag changes, a fire cycle is initiated. If a cooldown attribute was specified, the current cooldown status is ignored and also won't be reset. Defaults to nil (no flag watched)
cooldown	Used with Artillery UI: Number of seconds before the next fire cycle can be initiated. Is ignored when initiating fire via ME flags. Defaults to 120 (2 Minutes)
baseAccuracy	The radius (in meters) around the center of the zone in which the projectiles will land. Defaults to the ME zone's radius (meaning all projectiles will land inside the zone if this attribute is missing and fire cycle is invoked via trigger flag)
silent	Used with Artillery UI: if true, suppresses communication responses from artillery

Note that all zones that are created with ME are also automatically added to the pool of managed artillery zones.

### 3.4.7.5 API

In addition to configuring artillery zones with ME, mission designers can use the API for even finer control

#### 3.4.7.5.1 `addCallback(theCallback)`

Adds theCallback to the list of callbacks. Your callbacks will be invoked whenever a fire or impact event occurs.

#### 3.4.7.5.2 `createArtilleryTarget`

`createArtilleryTarget(name, point, coalition, spotRange, transitionTime, baseAccuracy, shellNum, shellStrength, shellVariance, triggerFlag, addMark, cooldown, autoAdd)`

Creates an artillery zone via API. For the description of the various parameters and their default, please refer to the ME Attributes section, below.

The method returns a `cfxArtilleryZone` that can be added to `cfxArtilleryZone`'s list of target zones that are managed (for watching trigger flags). If `autoAdd` is set to true, the newly created artillery zone is automatically submitted.

#### 3.4.7.5.3 `addArtilleryZone(aZone)`

Adds `aZone` to the list of managed artillery zones. Note that only zones added to artillery zones' list of managed zones can show up on the map or will be considered when querying `artilleryZonesInRange`

#### 3.4.7.5.4 findArtilleryZoneNamed(aName)

Returns the artillery zone with name aName, or nil otherwise. aName must be a full match.

#### 3.4.7.5.5 removeArtilleryZone(aZone)

Removes aZone from the list managed artillery zones. The zone will disappear from the map and no longer be considered by artilleryZonesInRange. Remove an artillery zone when you know that all targets inside the zone have been destroyed.

#### 3.4.7.5.6 artilleryZonesInRangeOfUnit(theUnit)

Returns a table of all artillery zones that fulfill the following constraints

- is managed by artillery zones
- belongs to the same coalition as theUnit
- theUnit is at maximum spotRange from the zone's center
- theUnit has LOS to the zone's center point

#### 3.4.7.5.7 doFireAt(aZone, maxDistFromCenter)

Initiates a fire cycle at the artillery zone aZone. maxDistFromCenter specifies the maximum distance the projectiles will land (i.e. the 'accuracy' of the shells). If omitted, the zone's baseAccuracy attribute is used.

doFireAt ignores any cooldowns, and will not reset the cooldown of an artillery zone.

aZone can be a string with the name of the artillery zone. aZone does not have to be actively managed by artillery zones to invoke doFireAt.

#### 3.4.7.6 Using the module

Include the cfxArtilleryZones source into a DOSCRIPT action at the start of the mission

Place artillery zones with ME

### 3.4.8 cfxOwnedZones

#### 3.4.8.1 Description

Owned Zones is a module that managers 'conquerable' zones that spawn attackers and defenders, and that keeps a record of which coalition owns which zone. Ownership is updated regularly. Owned Zones anchors itself to zones with an 'owner' attribute from ME.



#### Note

'owner' is an attribute that *all* DML Zones share: it is assigned implicitly by cfxZones, and set to neutral by default. It is only by **explicitly** setting an 'owner' attribute in ME that a zone becomes an Owned Zone.

#### Visuals

Owned zones are shown on the F10 Map in-game and are colored by their owning faction: grey for neutral, Red for REDFOR, Blue for BLUEFOR. This can be turned off for each zone by an attribute.



#### Conquering an Owned Zone

An Owned zone is conquered when there are only ground troops belonging to the opposing (conquering) faction inside the zone left alive. A single ground unit (including infantry) can therefore conquer a zone, as long as there are no units from the opposing faction inside the zone. Capturing a zone is instantaneous. A neutral zone is captured even if there are neutral units remaining in the neutral zone, i.e. neutral units do not have to be destroyed to capture a neutral owned zone.

#### Zone Protection Attributes

There are several attributes that can protect an owned zone from the enemy. You can use this to prevent certain conditions from arising (such as a critical owned zone is inadvertently taken out by AI instead of players).

- Owned Zones can be set to "unbeatable" so they are never conquered by another faction.
- Owned Zones can be set to "untargetable" so that AI will ignore them when looking for a zone to attack.

#### Defenders / Attackers Production Logic

Owned zones can spawn troops to defend the zone (defenders), and send out troops to engage other owned zones. What troops they produce are determined with the 'defendersRED/BLUE' and 'attackersRED/BLUE' Type attributes. Neutral zones do not produce attackers nor defenders.

The logic for production is as follows

- When the mission starts up, defenders for the currently owning faction are produced instantly, unless the zone is neutral, in which case no defenders are created.



- When a zone is captured, the zone enters a “defender production” cycle (it waits). If at the end of the wait cycle the zone is still held by the same faction, defenders are spawned as described by the defendersXXX property.
- Once all defenders are spawned, the zone goes into attacker production (wait) cycle.
- When no defenders get destroyed during the produce attacker cycle,
  - If there are no enemy or neutral zones to attack, the zone spawns no units.
  - A new attacker group (consisting of units as described in attackersXXX) is spawned that automatically seeks out other owned zones [requires cfxGroundTroops] that are owned by neutral, or the other faction.
  - A new attacker production wait cycle starts
- When a zone defender is destroyed, the zone enters a ‘shocked’ state in which it does nothing. This shocked counter is renewed every time defenders are destroyed. Once the shock counter finishes, the zone enters a repair cycle
- In repair, all damaged units are replaced by fresh ones one by one, one unit for each cycle. When all defenders are repaired, the zone goes back to producing attackers.

Note that once attackers are produced, the module attempts to hand them off to cfxGroundTroops with orders to “attackOwnedZone”. If cfxGroundTroops is not loaded, this results in an error message.

### ME Flag Integration

Owned Zones support ‘bang!’ attributes for red, blue and neutral: you can specify one flag (e.g. 100) for each side (red, blue, neutral) that Owned Zones changes each time an Owned Zone changes hands. The logic is that the flag for the winning side is increased by one, and the one for the side that lost the zone is decreased by one.

Name	Value	Description
n!	Number	Increase this flag by one if neutral wins an owned zone. Decrease this flag by one if neutral loses an owned zone
r!	Number	Increase this flag by one if red wins an owned zone. Decrease this flag by one if red loses an owned zone
b!	Number	Increase this flag by one if blue wins an owned zone. Decrease this flag by one if blue loses an owned zone

### Callbacks (Lua)

When a zone changes hands, a capture callback can be invoked. You install such a callback via:

```
function cfxOwnedZones.addCallBack(conqCallback)
```

conqCallback has the signature (zone, newOwner, formerOwner) with zone being the cfxZone, and newOwner and formerOwner the respective coalition ID (0 = neutral, 1 = red, 2 = blue)

#### 3.4.8.2 Dependencies

**Required:** dcsCommon, cfxZones

**Optional:** cfxGroundTroops

### 3.4.8.3 Module Configuration

To configure the Owned Zones module via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is off
announcer	Show a message if an owned zone is captured. Default is true
defendingTime	Time (in seconds) for producing defenders. Defaults to 100
attackingTime	Time (in seconds) for producing attackers. Defaults to 300
shockTime	Time (in seconds) after an attack on defenders before repair commence. Defaults to 200
repairTime	Time (in seconds) for repairs to complete. Defaults to 200
n!	Increase this flag by one if neutral wins an owned zone. Decrease this flag by one if neutral loses an owned zone
r!	Increase this flag by one if red wins an owned zone. Decrease this flag by one if red loses an owned zone
b!	Increase this flag by one if blue wins an owned zone. Decrease this flag by one if blue loses an owned zone

### 3.4.8.4 ME Attributes

Name	Description
owner	Coalition that owns the zone at beginning of Mission. Can be 0, 1, 2 or “red”, “blue”, “neutral”. If nothing or some illegal value give, this defaults to neutral (0) <b>MANDATORY</b>
defendersRED	A string, coma separated, that specifies the types of troops to spawn when the zone is owned by RED. Example: "Soldier M4,Soldier M4" places two Infantry soldiers. <b>Warning:</b> these types need to <i>exactly</i> match DCS's types. Be sure not to accidentally insert blanks. Special types: “none” – no troops Defaults to “none”
defendersBLUE	A string, coma separated, that specifies the types of troops to spawn when the zone is owned by RED. Example: "Soldier M4,Soldier M4" places two Infantry soldiers. <b>Warning:</b> these types need to <i>exactly</i> match DCS's types. Be sure not to accidentally insert blanks. Special types: “none” – no troops Defaults to “none”
attackersRED	A string, coma separated, that specifies the types of troops to spawn when the zone is owned by RED. Example: "Soldier M4,Soldier M4" places two Infantry soldiers. <b>Warning:</b> these types need to <i>exactly</i> match DCS's types. Be sure not to accidentally insert blanks. Special types: “none” – no troops Defaults to “none”
attackersBLUE	A string, coma separated, that specifies the types of troops to spawn when the zone is owned by RED. Example: "Soldier M4,Soldier M4" places two Infantry soldiers. <b>Warning:</b> these types need to <i>exactly</i> match DCS's types. Be sure not to accidentally insert blanks.

Name	Description
	Special types: “none” – no troops Defaults to “none”
formation	Formation of the defenders group. See dcsCommon for supported group formations. Defaults to ‘circle_out’.
attackFormation	Formation of the attackers group. See dcsCommon for supported group formations. Defaults to ‘circle_out’.
spawnRadius	Radius of circle that the defenders are placed on. Defaults to slightly less than zone radius, so defenders are always inside the zone they are defending. Defaults to 0.
attackRadius	Radius of circle in which the attackers spawn after they are produced. Defaults to zone radius
attackDelta	Distance from center of zone in which attackers spawn circle is located. Defaults to 10.
attackPhi	Angle (direction) in degrees from zone center where attackers are spawning. Defaults to 0.
paused	Pauses zone. “true” or “yes” means that the zone is paused. A paused zone produces no attackers nor defenders, but will detect capture normally. Capturing a paused zone will currently not unpause the zone. Do that in the capture callback. Defaults to “no”
unbeatable	“true” or “yes” makes it unbeatable. Zone can’t be conquered by other side. Defaults to “no”
untargetable	“true” or “yes” makes it untargetable. Zone will not be targeted by troops with ‘attackOwnedZones’. Defaults to “no”
hidden	“true” or “yes” hides it. Zone is not shown on F10 Map. Defaults to “no”

### 3.4.8.5 API

In addition to configuring owned zones with ME, mission designers can use the API for even finer control

#### 3.4.8.5.1 `addCallBack(conqCallback)`

Adds conqCallback to the list of callbacks the Owned Zones invokes when an owned zone changes hands. The callback must conform to the profile

```
function myCallBack(zone, newOwner, formerOwner)
```

#### 3.4.8.5.2 `getOwnerForZone(aZone)`

Returns the owner for DML zone aZone. Same as aZone.owner.

#### 3.4.8.5.3 `getEnemyZonesFor(aCoalition)`

Returns a table of all enemy owned zones for aCoalition. Note that this table is complete, it INCLUDES zones that have the untargetable attribute set to true.

#### 3.4.8.5.4 `getNearestOwnedZoneToPoint(aPoint)`

Returns the closest owned zone (any ownership) and distance to aPoint. It excludes owned zones that have the attribute ‘untargetable’ set to true

#### 3.4.8.5.5 `getNearestOwnedZone(theZone)`

Returns the closest owned zone (any ownership) and distance to theZone. It excludes owned zones that have the attribute 'untargetable' set to true

#### 3.4.8.5.6 `getNearestEnemyOwnedZone(theZone, targetNeutral)`

Returns the closest enemy owned zone (opposing coalition) and distance to theZone. If targetNeutral is true, neutral zones are included in the list. It excludes owned zones that have the attribute 'untargetable' set to true

#### 3.4.8.5.7 `getNearestFriendlyZone(theZone, targetNeutral)` excludes

Returns the closest friendly (same coalition) owned zone (any ownership) and distance to theZone. If targetNeutral is true, neutral zones are included in the list. It excludes owned zones that have the attribute 'untargetable' set to true

#### 3.4.8.6 *Using the Module*

To enable, add the script to the mission as a DOSCRIPT action during Mission Start

To configure the module, place configuration zone as described above.

Then, place Trigger Zones in ME, and name them. Add the 'owner' property and enter "red", "blue" or "neutral" as initial owners. All other properties are optional.

### 3.4.9 FARP Zones

#### 3.4.9.1 Description

FARPZones is a Zone Extension that improves in-game FARP capabilities when capturing FARPs: it automatically creates all units required to operate a fully functioning FARP (i.e. Power, Communication, Repair and Rearm), and can optionally also place defenders (similar to OwnedZones). The FARP Zone automatically reflects the owning status of the FARP object it is linked to (the FARP object must be inside the zone), and re-generate the resource/service vehicles once captured.

When creating a FARP zone, it is best to place it on, or very close to the center of the FARP object itself, so that the Resource Vehicles are easy to place, and you can ensure that the FARP is contained within the Zone.

FARPs are marked by a circle in the F10 player map, colored in the color of the owning faction.

Unlike OwnedZones, a FARP Zone currently does not re-generate defenders.

#### **Note:**

FARP Zones do not work with Airfields!

#### 3.4.9.2 Dependencies

**Required:** dcsCommon, cfxZones

**Optional:** -

#### 3.4.9.3 Module Configuration

Name	Description
spinUpDelay	Number of seconds after a capture that the FARP becomes active (the resource vehicles spawn). Example: 30

#### 3.4.9.4 ME Attributes

Name	Description
<b>FARP</b>	Indicates that this zone is a FARP zone. Value is ignored. <b>MANDATORY</b>
rPhiHDef	Radius (in m), Phi (degrees) and Heading (degrees) of the center point around which the defenders deploy. Defaults to 0, 0, 0
rPhiHRes	Radius (in m), Phi (degrees) and Heading (degrees) of the center point around which the resource vehicles deploy as a line. Defaults to 0, 0, 0
redDefenders	typeStrings of defender vehicles. Example "ZSU-23-4 Shilka, ZSU-23-4 Shilka". Defaults to "none" Special encoding: "none" – no vehicles
blueDefenders	typeStrings of defender vehicles. Example "Roland ADS,Roland Radar,Roland ADS". Defaults to "none" Special encoding: "none" – no vehicles
formation	Formation of the defenders group. See dcsCommon for supported group formations. Defaults to 'circle_out'.

Name	Description
hidden	Set to “no” if FARP is visible on the F10 map (and colored according to owner). Defaults to “no”
hideRed hideBlue hideGrey	For any of these three attributes, the FARP is hidden if it belongs to that faction. For example, if hideRed is set to true, the FARP is shown on the map while it belongs to neutral or blue, but disappears when it is owned by red.

#### 3.4.9.5 API

In addition to configuring FARP zones with ME, mission designers can use the API for even finer control

(tbd)

#### 3.4.9.6 Using the module

Add the script to your mission using a DOSCRIPT action while the mission starts.

In ME, place a FARP static object, and then a Zone over it (choose a radius of 2 km to match up with capture radius), and add the FARP attribute to the Zone.

### 3.4.10 cfxMapMarkers

#### 3.4.10.1 Description

A small ME extension module that allows you to place markers and text comments in ME on the map that players can see during the mission when they switch to F10 Map View (provided they enable markers).

#### 3.4.10.2 Dependencies

**Required:** dcsCommon, cfxZones

#### 3.4.10.3 Module Configuration

No special configuration required

#### 3.4.10.4 ME Attributes

Name	Description
mapMarker	Turns on the map marking feature. Simply must be present. Content of this property is displayed as text on the Map. Example "Destroy all vehicles in this area" <b>MANDATORY</b>
coalition	Side that sees this marker. Can be "red", "blue", "neutral", or "all". You can also substitute "1" for red, and "2" for blue. Defaults to "all"

#### 3.4.10.5 API

None.

#### 3.4.10.6 Using the module

To enable, add the script to the mission as a DOSCRIPT action during Mission Start



To use, place a Zone in ME, and name it. Then add the 'mapMarker' property and add descriptive text into the value field. That text is shown in-game on the F10 Map. All other properties are optional.

### 3.4.11 cfxNDB

#### 3.4.11.1 Description

This enhancement places an NDB (non-directional beacon) that aircraft can home in on with their ADF. Since cfxNDB is based on cfxZones, these NDB can move by linking them to a unit, making it easy to add 'homing beacons' to units that (in DCS) are difficult to add beacons to: ships. Look at the demo to see how we attached an NDB to a battlecruiser that a Huey can home in on.

NDB are exceedingly easy to set up – all they need is a frequency and a sound file (since DCS currently does not support a set of default sound files, you must supply your own. The 'ADF and NDB fun' mission includes a small (public domain) sound file you can use to simulate an ELT signal.

Name	Value	
NDB	121.5	
soundFile	distressbeacon.ogg	




To use an NDB in their aircraft, players must be familiar with radio navigation.

#### Moving NDB

If you use the linkedUnit attribute to make the zone follow a unit, an NDB will automatically observe any location change. In the example to the right, we have linked an NDB at 540 kHz to the naval unit named 'Cruiser.'



By default, a unit-linked (moving) NDB updates its location every 10 seconds. That is quite often, as most units do not move very far in that time (for example, the carrier "Theodore Roosevelt", when cruising at 50 km/h moves 140m in that time. That is less than half its length). ADF navigation isn't precise enough to notice small spatial changes unless very close by, so update (or 'refresh') intervals with longer times usually work equally well. Note that in order to reposition an NDB, the audio transmission (as defined by the sound file) is turned off and then re-started at the new location. This is important to remember if your refresh interval is shorter than the duration of the sound clip, as anything past the refresh interval is not played and the sound file begins anew. Location refresh is turned off for unlinked NDB.

Name	Value	
NDB	0.540	
soundFile	distressbeacon.ogg	
linkedUnit	Cruiser	

If required, you can change the update interval of NDBs with an attribute in the config zone

#### Sound File

The NDB transmits an endlessly repeating sound file over the radio. You must specify the sound file's name in the attribute, and include it's file type (e.g. ".ogg"). In order to work you must observe the following:

- The sound file must be included in the mission. The easiest way to do this is by adding a "Sound To All" Action that is timed at some point far in the future (some 99999 seconds after mission starts). This includes the sound file into the correct location in your mission.



- NDB looks for sound files in I10n/DEFAULT/. If you manually place sound files in your mission at other places than ME's default location ("I10n/DEFAULT/"), you must provide the path to that location yourself, relative to I10n/DEFAULT/.

#### 3.4.11.2 Dependencies

NDB requires dcsCommon and cfxZones.

It also requires that you include the sound files that you want the NDB to transmit.

#### 3.4.11.3 Module Configuration

To configure the NDB module via a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it "ndbConfig" (note: name must match exactly)
- Add any of the following attributes to this zone:

Name	Description
verbose	Show debugging information. Default is off
ndbRefresh	Time (in seconds) between location updates <i>for moving NDB</i> (i.e. an NDB with a linkedUnit attribute).  <b>Note</b> if the refresh interval is shorter than the duration of the sound file that is transmitted, the sound file stops playing at refresh, and then starts at the new location <i>from the beginning</i> . This means no part of the sound file beyond the refresh interval is ever played.  NDB that aren't linked to units do not refresh and have no restrictions on the length of their transmission

#### 3.4.11.4 ME Attributes

Name	Description
<b>NDB</b>	Creates an NDB at the zone's center. If the zone is linked to a unit, this NDB will automatically update to the unit's location.  <b>The value of this attribute is the frequency (in MHz) at which the NDB transmits (e.g. 121.5 for 121.5 MHz, 0.42 for 420 kHz)</b>  <b>MANDATORY</b>
fm	If true, the transmission is in FM, else in AM Defaults to false (AM)
ndbSound	Name of the sound file with extension that is to be transmitted. Defaults to '<none>'. Note that the sound file's name must be specified relative to the missions default location for sound files (I10n/DEFAULT/). If you use ME to import the sound files, you do not have to specify the location.
watts	Transmission power (in watts) for the NDB. 100 Watts usually has a range of some 150 km. Defaults to 100 Watts

#### 3.4.11.5 API (none)

#### 3.4.11.6 Using the Module

To enable, add the script to the mission as a DOSCRIPT action during Mission Start

To use, simply add the 'NDB' and 'soundFile' attributes to a zone.

### 3.5 Using Stand-Alone Features

DML stand-alone features are drop-in modules that out-of-the-box provide specific capabilities without requiring additional set-up (although mission designers can curtail these modules to their requirements with configuration- and data zones).

DML has ready-made modules that implement the following abilities (please note that for some modules documentation is forthcoming)

- **Player Score**  
A simple score board based on players (not units) for a more action-oriented approach to missions. Allows mission-specific score tables based both on unit type and unit name.
- **Helo Troops**  
Allows players in transport helicopters (Huey, Hip, Hind) to pick up infantry anywhere and deploy them somewhere else. Supports comprehensive in-game UI and closely works together with spawn zones (if present)
- jtacGrpUI (doc coming)  
Allows players to interact with DML-spawned JTAC troops that have 'lase' orders. Players receive vectoring upon request.
- CSAR Manager (doc coming...)  
Allows instant creation of CSAR missions with integrated pick-up of unit by landing close to them or simulated winching. Updates cargo weight.
- Limited Airframes (doc coming...)  
Very unfortunately named module, that facilitates a limited pool of player pilots. If a pilot is killed, ejects or ditches an airframe outside of designated safe zones, that pilot is lost. Integrates with CSAR Manager to automatically generate CSAR missions to recover downed pilots and re-plenish the player pilot pool.
- **Guardian Angel**  
A module that prevents missiles from hitting protected air units by destroying them shortly before impact. Supports comprehensive text-based missile warning.
- **Parashoo**  
A tiny module that gets rid of those pesky parachute guys that litter the ground after some time
- Civ Air (doc coming...)  
Generates civilian (neutral, transport) air traffic that flies between air fields. Aircraft spawn on the airport at ramp, start up, fly to their destination airport, land, taxi to ramp, and despawn after some time.
- Artillery UI (doc coming...)  
In-game player UI for DML Artillery (Target) Zones to allow players to activate the fire

command for artillery zones. Supports the artillery zone's spot range and LOS requirements

- **Recon Mode**

An airborne scouting/recon system that marks discovered groups on the map and supports priority- and black lists

- **SSB Client**

An advanced slot blocking system that can block slots for aircraft on airfields/FARPs that do not belong to the same side. Also supports "single-use" of aircraft (blocking a slot after a crash). There is also a stand-alone version of SSB Single-Use available.

- **SSB Single Use (coming...)**

A slot blocking mechanism that blocks aircraft after they have crashed.

- **cfxmon Development Tool (Lua Only)**

A development tool that mission designers using Lua can use to view all events (DCS, DML) as they occur

### 3.5.1 Player Score

#### 3.5.1.1 Description

Player Score is a module that automatically keeps score and a “kill log” for each player. Mission designers can add a unit score table for both unit types (e.g. a BTR-80 kill yields 20 points) or named units (the unit named “SAM Command West” yields 50 points. Mission designers provide a score table by adding a specifically named trigger zone, and then add the type- and name scores as attributes

Scoring is automatic and the score is kept for the player (by name), not the unit(s) they control. So, if a player changes air frames, the kills in the new frame is added to those they made before. Player Score itself keeps score for every kill and announces them to all players as they happen. A separate module “Player Score UI” provides a UI to access totals and kill log.

#### Announcing Kills

Player Score announces each kill with the score and current total for each side. This feature can be turned off with an attribute in a config zone. AI kills are only announced to the side that has earned a kill. PvP kills are announced on *both* sides.

Killing a named unit (i.e., the unit’s name is listed in the playerScoreTable) is announced as having successfully killed a strategic unit.

After each kill, the total score for the player that earned the kill is announced for their side.

#### Tubulating Kills Types

Internally, Player Score keeps a record of how many unit types (e.g. BTR-80) a player has killed. This information can be accessed by other modules (e.g. Player Score UI) or other scripts.

#### Named / Typed Scores

A kill for a unit that is mentioned on the playerScoreTable (either the unit’s name or its type) yields that score, and twice that amount if it was a fratricide. Name score has precedence over Type. For example, if the playerScoreTable has an entry for BTR-80 that yield 35 points, a kill of that unit type scores 35 points. If that unit was named “Field Commander” and the playerScoreTable has an entry of, for example, 100 points for “FieldCommander”, those 100 points are awarded for the kill instead of 35.

If the unit killed isn’t mentioned on the playerScoreTable (neither unit name nor type), a default score is used.

#### Default Scores

Default scores (unless changed via a config zone) are as follows:

- Aircraft: 50
- Helicopter: 40
- Ground Unit: 10
- Ship: 80
- Train: 5

## Sounds

Mission designers can supply two different sounds to be played: a sound that is played when a normal kill is scored, and a 'bad' sound that is played when a fratricide occurs, or a player is killed (played only on the side the killed player belongs to)

The name for the sound files is provided via the config zone.

## Support for scripts (Lua only)

Player Score supports a simple interface to fetch a player's score, and the ability to change a player's score by an amount. Please see the API section.

### 3.5.1.2 Dependencies

Player Score requires the modules dcsCommon, cfxZones and cfxPlayer.

### 3.5.1.3 Module Configuration

Player Score uses two different sources of data for configuration: a standard configuration zone for setting up how Player Score behaves, and a score table (data) zone where a mission designer can assign score for unit types (e.g. BTR-80) and individual units (by unit name).

## Configuration Zone

To configure Player Score module via a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it "playerScoreConfig" (note: name must match exactly)
- Add any of the following attributes to this zone:

Name	Description
verbose	A value of "true" turns on debugging messages. Default is "false"
aircraft	The fallback score to award for killing an aircraft if that unit wasn't found on the score table (name or type). Defaults to 50
helo	The fallback score to award for killing a helicopter if that unit wasn't found on the score table (name or type). Defaults to 40
ground	The fallback score to award for killing ground unit if that unit wasn't found on the score table (name or type). Defaults to 10
ship	The fallback score to award for killing a ship if that unit wasn't found on the score table (name or type). Defaults to 80
train	The fallback score to award for killing a train if that unit wasn't found on the score table (name or type). Defaults to 5
announcer	If false, no kills are announced. Score is still kept. Defaults to true
scoreSound	Name of the sound file to play when a score is announced
badSound	Name of the sound file to play when killing own troops or being killed in PvP

## Score Table

The score table holds entries for two different purposes: Score for types, and score for named units.

Whenever a unit is killed, Player Score first checks if the score table has an entry with the exact name as the unit's name (e.g. "SAM Commander" – the name given by a mission designer to a unit) and uses that score. If no match is found, it looks at the type of the unit that was killed (e.g. BTR-80) and tries to find a matching entry in the score table. If a match is found, that score is used. If no match for name nor type is found, the matching category (aircraft, helicopter, Ground, Ship, Train) from the config (see above) is used as score.

To use a score table in your mission,

- Place a Trigger Zone in ME anywhere
- Name it "playerScoreTable" (note: name must match exactly)
- Add names/types and their score to the table:

The score table uses the following format:

Name	Description
<type or name>	<Score as number>
Type Exampe: BTR-80	Example: 15
Name Example: Big Kahuna	Example: 130

### 3.5.1.4 ME Attributes

None. (not counting configuration / data zones, see above)

### 3.5.1.5 API

3.5.1.5.1 [updateScoreForPlayer\(playerName, score\)](#)

3.5.1.5.2 [logKillForPlayer\(playerName, theUnit\)](#)

3.5.1.5.3 [scoreTextForPlayerNamed\(playerName\)](#)

### 3.5.1.6 Using the module

Copy the script into a DOSCRIPT action while the mission starts.

Add a score table or config zone if you want to assign other scores than the default values.

## 3.5.2 cfxHeloTroops

### 3.5.2.1 Description

HeloTroops adds the ability to Airlift (transport: load / unload) ground troops into transport helicopters. It installs an “Airlift Troops...” command into a player’s Communication→F10 Other... menu to allow them to load, unload, and set troop transport preferences.

Note that Helo Troops can load any group that complies with Helo Troop’s ‘legalTroops’ unit filter (infantry only by default, can be customized in a config zone). Having these units spawn with DML is not a precondition.

When flying a transport helicopter (as defined in dcsCommon), the script loads and deploys troops when the helicopter is on the ground. When close to a cfxSpawnZone with ‘requestable’ attribute, it can also trigger a spawn. Landing close to any group that entirely consists of transportable troops (as defined in HeloTroops), allows the player to load these troops into the helicopter for transport.

If the helicopter lands and has troops loaded, these troops can be (auto-)deployed. The script supports user-configurable settings to auto-load the closest loadable group when no troops loaded, and auto-deploy any loaded troops when troops are being carried. This enables the player to immediately deploy any loaded troops on touch-down.

Currently, the player preferences default to auto-load = OFF and auto-deploy = ON (can be changed with a config zone).

### Helo-Troops built-in UI

Helo Troops provides a UI via Communication→Other...→Airlift Troops menu that allows players to

- Request spawns from spawners in range (will start a spawn cycle on that spawner. Checks for cooldown first)  
Note: if there are more than 5 spawners in range, only the first 5 are shown)
- Load troops into the helicopter (Choose by team in range. List is limited to the closest 5 teams)
- Deploy troops loaded in the helicopter
- Change Auto-deploy and Auto-load settings

### Interaction with DML Modules

Helo Troops automatically interacts with the following modules if they are present:

- SpawnZones – find and interact with spawner in proximity to the helicopter to their start spawn cycle upon request (player-controlled)
- GroundTroops – manages ‘wait-’ prefix and removes that prefix when deploying troops that had orders with ‘wait-’ prefix upon loading
- CSAR Manager – compatible with UI and loading of downed pilots
- Cargo manager (tbc) – weight management

### Other

HeloTroops fully supports multi-player; in MP, player groups **must** be single-unit or the scripts will not work correctly.

Currently, the script does not change the helicopter’s cargo weight. This feature is expected to be added soon.



### 3.5.2.2 Dependencies

**Required:** dcsCommon, cfxZones, cfxCommander, cfxGroundTroops

**Optional:** cfxSpawnZones (for requestable troop spawning)

### 3.5.2.3 ME Attributes

None.

### 3.5.2.4 Module Configuration (tbc)

To configure the Helo Troops module via a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it "heloTroopsConfig" (note: name must match exactly)
- Add any of the following attributes to this zone:

Name	Description
legalTroops	Type Array that identifies the unit types that helicopters can load. This is compared against any unit on the ground to determine if the helicopter can load the group. All units in the group must be on that list, or the entire group cannot be loaded. For example, if a group consists of four infantry soldiers, the group can be loaded. If the group also contains a vehicle (e.g. "Hummer"), that group cannot be loaded. Defaults to "Soldier AK, Infantry AK, Infantry AK ver2, Infantry AK ver3, Infantry AK Ins, Soldier M249, Soldier M4 GRG, Soldier M4, Soldier RPG, Paratrooper AKS-74, Paratrooper RPG-16, Stinger comm dsr, Stinger comm, Soldier stinger, SA-18 Igla-S comm, SA-18 Igla-S manpad, Igla manpad INS, SA-18 Igla comm, SA-18 Igla manpad"
troopWeight	Used to calculate the cargo weight per troop loaded. Currently not used. Defaults to 100 (kg)
autoDrop	Default setting for helicopter when touching down. Players can change this individually. Defaults to true
autoPickup	Default setting for helicopter when touching down. Players can change this individually. Defaults to false
pickupRang	Range in which troops can be picked up, from helicopter. Defaults to 100 meters

### 3.5.2.5 API

None.

### 3.5.2.6 Using the module

Add the script to your mission using a DOSCRIPT action while the mission starts. All transport helicopters now can transport infantry.



### **3.5.3 jtacGrpGUI**

3.5.3.1 *Description*

3.5.3.2 *Dependencies*

3.5.3.3 *Module Configuration*

3.5.3.4 *ME Attributes*

3.5.3.5 *API*

3.5.3.6 *Using the module*

### 3.5.4 csarManager

#### 3.5.4.1 Description

csarManager is an extension for DCS Missions that adds CSAR (Combat Search And Rescue) capabilities to missions. CSAR missions can either be added to a mission via CSAR-Zones in ME that are generated at mission start, or via other scripts/modules (e.g. limitedAirframes) that generate CSAR missions at runtime. CSAR missions are only available to troop transport helicopters (i.e. Huey, Hind, Hip – not Shark, Gazelle nor Apache)

When using CSAR Zones, this script creates an entry into the managed missions table, and generates the required troops on the ground. Optionally (depending on the zone Attributes) the unit(s) also broadcast an emergency signal to home in on with ADF. Other scripts can generate a CSAR mission at any time by invoking

```
csarManager.createCSARforUnit(theUnit, pilotName, radius, silent)
```

In order to function properly, each side that has CSAR mission must have at least one CSARBASE defined where a helicopter can drop off rescued personnel. Without a CSARBASE, helicopters can pick up and drop off downed pilots, but the CSAR missions do not register as complete. When landing in a CSARBASE zone, any loaded rescued troops are automatically unloaded, the mission marked as successful, and any registered success callbacks are invoked.

To register a success callback, use

```
function csarManager.installCallback(theCB)
```

with the callback having the following signature:

```
function cb(theCoalition, success, numRescued, notes)
```

with

- `theCoalition` being a number for the coalition the helicopter that completes the mission belongs to
- `success` a bool if the CSAR mission was successful (true) or not
- `numRescued` the number of people rescued
- `notes` a string

Note that a callback is invoked separately for each mission. If a pilot picks up multiple evacuees from different CSAR missions prior to returning, and then lands at a CSARBASE, a success callback is invoked for each mission completed.

The script supports routing a player to CSAR mission targets and 'live update' during hover. For each active mission, a pilot can query via communications the target's bearing, range, and ADF frequency.

Picking up evacuees is handled automatically by landing in close proximity, or hovering at 3+ meters (9 feet or more) directly over the target for the required number of seconds, while not exceeding a maximum altitude. Pilots can pick up multiple evacuees before returning them.

The script correctly manages weight for any units picked up/dropped off.

csarManager can handle multiple active CSAR missions, and is fully MP capable. In MP, player groups must be single-unit groups.

CSARBASE works in conjunction with other cfxZone attributes like "FARP".

#### 3.5.4.2 Dependencies

**Required:** dcsCommon, cfxZones, cfxPlayer, nameStats, cargoSuper

**Optional:** limitedAirframes

#### 3.5.4.3 Module Configuration (tbc)

Name	Description
useSmoke	When approaching a mission target, activate smoke or not. Smoke can have significant performance impact when close in a helicopter, so the
rescueRadius	Helicopter must land within this distance (in meters) to the target to pick up. Recommended Value: 70
hoverRadius	When attempting a hover rescue, helicopter must stay within this range (in meters). Recommended value: 30
hoverAlt	When attempting a hover rescue, helicopter must stay below this altitude (in meters). Recommended value: 40
rescueTriggerRange	When approaching a mission target, the mission triggers a message from the evacuees at this range. This is also the range at which smoke is triggered if enabled
beaconSound	Name of sound file (ogg or wav) to play on the ELT frequency. Includes extension. Example: "Radio_beacon_of_distress.ogg"
pilotWeight	Weight for an evacuee in kg. Recommended Value: 100
hoverDuration	Time required to hover above pilot to secure winch and complete rescue

#### 3.5.4.4 ME Attributes

csarManager uses two different kinds of zones that you can place in ME to accomplish different things: a CSARBASE marks locations where pilots deliver the people they rescued; CSAR Zones are used to place pre-made CSAR mission on the map that are available at mission start.

##### 3.5.4.4.1 CSARBASE

A CSARBASE is a zone in which a helicopter transporting evacuees can unload the rescued personnel.

Name	Description
<b>CSARBASE</b>	Must be present to identify this zone as CSAR Base where CSAR Missions can end. A helicopter must land inside this zone. Supports linked zones (for example if the BSAR Base is a ship). Each side that has CSAR Missions must have at least one such zone, or CSAR Missions can not be completed. There is no upper limit on the number of CSAR Bases a side can have. The value of this attribute can be used to name the CSAR Base, else the Zone's name is used. <b>MANDATORY</b>

Name	Description
coalition	The side that owns the CSAR Base. If neutral, both sides can use this as a base, else only the faction specified. Defaults to “neutral”. Other possible values are “red” and “blue”
name	Optional name for CSARBASE.

#### 3.5.4.4.2 CSAR Zone

A CSAR Zone is a zone that allows you to place CSAR missions on the map. Upon mission start, these are picked up by the csarManager, and converted into active CSAR missions.

Name	Description
<b>CSAR</b>	Identifies this as CSAR Zone that is converted into a CSAR mission upon mission start. The size of this zone in ME is not relevant <b>MANDATORY</b>
coalition	Faction (red/blue) for which this mission is generated
name	Name of this mission, recommended is to use a personal name, e.g. “Lt. Wesley Crasher”
freq	Frequency for the ELT (radio to home in on) in KHz. Random if not set
timeLimit	(currently not used)
weight	Weight of pilot (tbc)

Note: need to complete CSAR weight

#### 3.5.4.5 API

csarManager allows you to generate CSAR missions while the mission is running. Invoke

```
csarManager.createCSARforUnit(theUnit, pilotName, radius, silent)
```

with the following parameters to create a new CSAR Mission

- `theUnit` is DCS unit the unit that ‘creates’ the CSAR mission, i.e. the unit that is crashing and where people are bailing out from. It must exist and have a location, but does not have to be alive. This unit is used to determine the CSAR mission’s location and faction (the CSAR mission is created for the same side that the unit belongs to)
- `pilotName` is a string that is used to create the mission’s name. It helps if it’s a good name, and the word (downed) will be prepended for creating the mission name. If you don’t give a name, ‘Eddie’ (as in ‘Eddie the Eagle’) is used.
- `radius` is the maximum distance from theUnit’s location where the mission is going to be created. This simulates the ‘parachuting to ground’. Use a fixed value, or a combination of altitude and speed to create some realistic randomized location.
- `silent` – a bool you can use to suppress the ‘Mayday’ message that is automatically generated when this message is invoked. Defaults to false

Note that not all invocations of this method result in CSAR missions: if the mission’s location (after randomizing from theUnit’s location) ends up in water, the units are assumed to have drowned. If not silent, a “KIA” message is displayed.

#### 3.5.4.6 Using the module

Add the script to your mission using a DOSCRIPT action while the mission starts.

In ME, place CSARBASES:

- CSARBASESs are zones where a helicopter can deliver rescued units to complete CSAR missions. Without a CSARBASE for their fraction, pilots can't complete a CSAR mission
- If you place a CSARBASE without a coalition attribute, or set the attribute to neutral, any player can complete a CSAR mission there
- (Optional) Place CSAR zones on the map for DCS to pick them up on start, and convert CSAR Zones to active CSAR missions that are available immediately
- To create CSAR missions while the mission is running, see the API section

### **3.5.5 Limited Airframes (tbc)**

3.5.5.1 *Description*

3.5.5.2 *Dependencies*

3.5.5.3 *Module Configuration*

3.5.5.4 *ME Attributes*

3.5.5.5 *API*

3.5.5.6 *Using the module*



### 3.5.6 Guardian Angel

#### 3.5.6.1 Description

Guardian Angel is a module that watches an aircraft and can protect it from incoming guided missiles. When a missile is fired at a protected unit, guardian angel first warns the unit, and then tracks, and 'intervenes' shortly before the missile hits by destroying it. Both warnings and interventions are optional and can be turned off.

**Warning:** Guardian Angel does *not* protect units against dumb-fire missiles nor guns.

Use this module to selectively make units (nearly) impervious against missiles, to add heart-attack-inducing segments to a mission (when the player does not know a guardian angel is watching them) or to create smart missile defense trainers.

Guardian Angel has the following features (most of which can be controlled with a configuration zone):

- Automatically protects player aircraft (fixed-wing and rotor-wing)
- Warns of missile launch with direction
- Destroys missiles shortly before they hit a protected unit
- Can announce warnings and 'interventions' to all or privately to the unit only
- Can protect player and AI planes
- Can add visual effects (small explosions) when a missile is removed (can be dangerous!)

Guardian Angel's missile protection is quite impressive. Run the demo mission "missile evasion (Guardian Angel)" to see how it can protect you and a fellow protected (AI) plane from multiple SA-6, S-10, and S-11 sites – while the other AI planes all get shot down.

Out of the box, Guardian Angel provides full protection for all player planes against all missiles. Using a config zone, you can selectively turn off some of the above mentioned features. For example, by turning off interventions, pilots are warned when a missile is launched, but they are no longer saved from the missile: a good and lethal training tool (with only interventions off, pilots are still informed when a missile has missed, lost track or re-acquired. A hit announces itself).

#### WARNINGS

Next to Guardian Angel's ability to protect planes from fiery missile death, it provides some comprehensive warnings that a RIO may give you:

Missile, missile, missile, 12 o clock
Missile, missile, missile, 6 o clock
16800000: tracking Froghopper, d = 9497m, Vcc = 157m/s, LR= 28mMissile MISSED!
16800000: tracking Froghopper, d = 9585m, Vcc = -12m/s, LR= 2mMissile RE-ACQUIRED!
16799232: tracking Froghopper, d = 205m, Vcc = -1272m/s, LR= 228m ANGEL INTERVENTION

#### Possible Warnings

Except for launch, all warnings begin with some 'gibberish' and end on the actual Warning. Let's look at the gibberish first, as you can use it later to gauge your own skills:

16799488: tracking Froghopper, d = 102m, Vcc = -747m/s, LR= 134m

The information displayed is as follows:

<weapon name> tracking <target unit name> <dist> <vcc> <LR>

with

- <weapon name> being whatever name DCS gave that thing. Yes, they are mostly uninspiring names.
- <target unit name> is the name of the unit that the weapon is fired at
- <dist> is the distance from target to the missile when the event occurred
- <vcc> is closing velocity between target unit and missile. A negative value means that the missile is moving closer, a positive that the missile is moving away
- <LR> is the resulting lethal range, and calculated by Guardian Angel based on closing velocity.

And now for the Warnings:

- *Missile, missile, missile*  
A missile is launched. Always comes with a clock direction
- *Missile Missed*  
A missile no longer closes in on the aircraft
- *Missile Re-Acquired*  
A missile re-gained track / closes in again
- *Missile Lost Track*  
A missile is no longer tracking the aircraft
- *Missile Disappeared*  
A missile was destroyed by other means than Guardian Angel
- *Angel Intervention / God Intervention*  
A missile was destroyed by Guardian Angel because it would have hit within the next 0.1 seconds.

### Callbacks (Lua Only)

Guardian Angel supports callbacks so scripts can be informed when angels intervene. A callback must match the following profile:

```
function angelCB(reason, targetName, weaponName)
```

When invoked, reason contains the `reason` for the invocation, `targetName` and `weaponName` as strings. Currently, the following reasons are defined:

- "launch"  
A missile was launched
- "miss"  
A missile has apparently missed the target. Re-acquisition can not be ruled out (and is likely for more advanced SAM)

- “reacquire”  
A missile that looked as if it missed / did no longer track is again tracking
- “trackloss”  
A missile has apparently lost track of the target. Re-acquisition cannot be ruled out.
- “disappear”  
A missile has disappeared – this is only invoked for missiles that Guardian Angel tracked and did not remove itself. Most likely reason is that the missile was destroyed.
- “intervention”  
Guardian Angel has removed a missile that was about to kill a protected unit

Your script can sign up to be invoked with

```
guardianAngel.addCallback(theCallback)
```

### 3.5.6.2 Dependencies

Guardian Angel requires dcsCommon and cfxZones

### 3.5.6.3 Module Configuration

Guardian Angel can use a configuration zone for setting up main options. To configure this module via a configuration zone,

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

Name	Description
verbose	A value of “true” turns on debugging messages. Default is “false”
autoAddPlayer	When set to true, player planes are automatically added to Guardian Angel’s watchlist. Default is true
launchWarning	If true, Guardian Angel announces a missile launch. Default is true
intervention	If true, Guardian Angel destroys a missile before it destroys a watched aircraft. Default is true
announcer	If set to false, Guardian Angel suppresses all announcements. Defaults to true
private	If set to true, all announcements are only made to the group that a missile was fired at. Set to false (everyone can see)
explosion	Guardian Angel can add a mostly harmless explosion when a missile is removed due to an intervention. If this value is smaller than one (e.g. -1) this feature is turned off. If you enter a value > 0 (zero), an explosion with a magnitude of this value is placed in direction of that missile’s last location, 500m from the aircraft. A mostly harmless value is 1.0 (one point zero)

Name	Description
	<p><b>WARNING I</b> Even though this explosion is usually harmless for the protected plane, it can pose lethal to any other plane (wingmen).</p> <p><b>WARNING II</b> The explosive effect is only harmless to the protected plane if the explosion value is small (e.g. 1). If you enter sufficiently larger values, the shock wave can destroy even the protected plane.</p> <p>If you set this value to see explosions, make the value 1.0</p> <p>Defaults to -1 (off)</p>

#### 3.5.6.4 ME Attributes

None.

#### 3.5.6.5 API

Guardian Angel has a simple API for interfacing with ME to add and remove AI planes to the watchlist. It also supports callbacks for Angel Events

##### 3.5.6.5.1 `addUnitToWatch(aUnit)`

Adds aUnit to the list of units that are to be watched (protected). aUnit can be a string containing the unit's name or a unit. This is primarily useful to add AI units with a ME action

##### 3.5.6.5.2 `removeUnitToWatch(aUnit)`

Removes aUnit from the list of watched units. If aUnit doesn't exist or isn't watched, this is ignored. aUnit can be a unit or string containing the unit's name

##### 3.5.6.5.3 `addCallback(theCallback)`

Adds theCallback to the list of methods to invoke when an angel event happens.

#### 3.5.6.6 Using the module

Include the guardianAngel source into a DOSCRIPT Action at the start of the mission

Optionally, add a config zone with ME

### 3.5.7 parashoo

When planes in DCS are shot down, their pilots can try to eject. If they eject successfully, they glide to the ground, and an icon with text appears on the F10 map to mark the landing spot.



Unfortunately, there is no way to interact further with these downed pilots, and the icons can start cluttering up the map in long engagements

#### 3.5.7.1 Description

parashoo is a simple, lightweight script that removes a parachutist a short while after they land on the ground.

#### 3.5.7.2 Dependencies

None. This script is stand-alone and can be added to any mission without requiring any other scripts.

#### 3.5.7.3 Module Configuration

`parashoo.killDelay` controls the time delay between the moment that the parachutist touches down, and the unit is removed. Default delay is 3 minutes.

#### 3.5.7.4 ME Attributes

None.

#### 3.5.7.5 API

None.

#### 3.5.7.6 Using the module

Copy the parashoo source into a DOSCRIPT action that runs at the start of the mission

### **3.5.8 Civ Air**

3.5.8.1 *Description*

3.5.8.2 *Dependencies*

3.5.8.3 *Module Configuration*

3.5.8.4 *ME Attributes*

3.5.8.5 *API*

3.5.8.6 *Using the module*

### 3.5.9 Artillery UI

#### 3.5.9.1 Description

Artillery UI is a smart interface for artillery zones that can guide players to artillery zones and gives them the ability to mark and fire artillery zones via the Communication→Other menu.

This UI is usually only intended for helicopters, but the UI can be made accessible to all aircraft (Note: due to proximity that is required for an aircraft to function as FO, using a fixed-wing aircraft as FO makes little sense)

Artillery UI directly interfaces with artillery zones and thus provides a drop-in command interface for players to control artillery zones. Artillery UI provides information and command via the Communication→Other... interface as follows

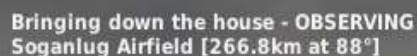
#### FO Rules that Artillery UI automatically observes – and how to get around them

Before Artillery UI allows a player to trigger the fire cycle of an artillery zone, the following conditions must be met:

- Player **must be in a helicopter** (unless the `allowPlanes` attribute is set to `true` in Artillery UI's config zone. In that case, every player unit has access to Artillery UI)
- Player must be **inside** an artillery zone's **spotRange** (unless the `allRanging` attribute is set to `true` in Artillery UI's config zone. In that case, all players have unlimited spotRange). Note that spotRange is an attribute of the individual artillery zones and can be edited with ME
- **Player's** view to the target's center is unobstructed and they have a **LOS** (unless the `allSeeing` attribute is set to `true` in Artillery UI's config zone. In that case, they always have unobstructed view)
- The artillery zone's **cooldown** timer has run out (unless the `allTiming` attribute is set to `true` in Artillery UI's config zone. In that case, the cooldown is reduced to zero). Note that cooldown is an attribute of the individual artillery zones and can be edited with ME

#### Target Direction / Guidance

Artillery UI provides a list of all artillery zones currently managed by the artillery zones module. If the group querying target directions is further than a few kilometers (the zone's spotDistance, to be precise) away from a zone, the list includes bearing and range to the target.



If the unit is close enough to observe the target zone, OBSERVING is reported instead for that target zone. If the unit is in range, but the player has no LOS to the target zone, "OBSCURED" is reported.

When a target is reported as OBSERVING

#### Marking Zones

Artillery UI allows players to request target zones to be marked. Instead of artillery shells, a

single phosphorous round is shot into the target zone, marking the zone visually with colored smoke. Smoke dissipates after 3-5 minutes

### **Fire Control**

When a unit is close enough to observe the target zone, and has a direct line of sight (LOS) to the target zone's center, the unit can order the artillery to fire. Note that is usually is this requirement (close proximity and LOS to the artillery zone's center) that makes it next to impossible for modern fighter aircraft to be effective at FO: their time over target is simply too short.

### **Reload in Artillery**

After firing into an artillery zone, the artillery needs to re-load. This takes time (as configured with the artillery zone's cooldown attribute which defaults to two minutes). Fire commands into the artillery zone before that time are ignored.

#### *3.5.9.2 Dependencies*

Artillery UI requires the following modules: dcsCommon, cfxZones, cfxPlayer, cfxArtillerZones

#### *3.5.9.3 Module Configuration*

ArtillerUI can use a configuration zone for setting up main options. To configure this module via a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it "ArtilleryUIConfig" (note: name must match exactly)
- Add any of the following attributes to this zone:

Name	Description
verbose	A value of "true" turns on debugging messages. Default is "false"
allowPlanes	Usually, the Communication menu is only visible in helicopters, as fixed-wing aircraft can't loiter close enough to artillery zones to act as FO. Setting this to true also gives fixed-wing aircraft access to the artillery UI. Defaults to false
smokeColor	This defines the smoke color used to mark artillery zones. Defaults to "red". Legal values are "green", "blue", "orange", "red", "white" and the numbers 0 through 4
allSeeing	Removes the unobstructed view requirement for all players. They now can fire when in range. Default is false
allRanging	Removes the spot range requirement for all players. They now are always in range. Default is false
allTiming	Removes the cooldown restriction for all players. Artillery zones can always start a new fire cycle. Default is false

#### *3.5.9.4 ME Attributes*

None.



#### 3.5.9.5 *API*

None.

#### 3.5.9.6 *Using the module*

Include the cfxArtilleryUI source into a DOSCRIPT Action at the start of the mission

Optionally, add a config zone with ME

### 3.5.10 Recon Mode

#### 3.5.10.1 Description

Recon Mode allows 'scout' planes (AI and Player) to automatically record enemy ground troops on the F10 map that then become visible to all players on the same side to see. This is similar in principle to DCS's built-in 'fog of war' map feature, but has several important differences:

- recon ability can be assigned to and removed from specific planes
- supports a 'priority list' – units that are a priority to find for the recon planes
- supports a 'black list' – units that recon planes never find
- sports callbacks so your own scripts can tap into recon results
- sports ME flag integration for when a scout detects units and priority targets
- detection is based on configurable parameters (altitude and visibility)
- can announce detections
- can mark detected units on the F10 map



#### Announcements

Whenever a scout detects an enemy unit, an announcement is made to all players of the scout's side. This feature can be turned off

#### F10 Map Marks

Whenever Recon Mode detects an enemy unit, it places a mark on the F10 map that all players on the same side can see. The mark contains some additional information (how many units sighted, group name). The mark does not update, and therefore represents the initial contact location and strength. The mark remains is automatically removed after 30 minutes (or when any player clicks on the "X" icon in the mark's description.). This feature can be turned off with a config zone

#### Detection Range

Auto Recon can detect units at far greater ranges than they are in DCS (with Fog of War set). Detection range is a function of two user-configurable (via a config zone) attributes: minimum- and maximum range. The unit's actual detection range is a function of altitude (above ground). When close to the ground, detection range is at minimum, and when at high altitude at maximum.

## Performance Considerations

A recon function has to regularly check detection against all existing troops on the ground. This can quickly escalate in terms of performance requirements. Recon Mode uses a number of methods to intelligently limit its performance drain on the mission:

- Recon planes are kept at minimum
- Recon planes check every few seconds, not permanently
- Recon checks are spread over time, not all at once
- Recon never reports neutral troop contacts
- Recon mode makes some assumptions with regards to how groups are organized and further reduces performance drain

In short, when under pressure, Recon Mode trades detection accuracy for performance: instead of hitting your CPU up for more power, it relaxes the recon schedule. The result is that a recon plane may be a few seconds late in reporting a new contact.

With those automatic limitation in place, Recon Mode reduces performance impact to negligible levels even if you have thousands of units on the map. You can therefore use recon mode in large-scale multi-player missions without worrying about Recon Mode dragging performance down.

## ME Integration

You can set up Recon Mode to increase a flag every time it detects an enemy unit, and a different flag every time it detects a priority target (see Priority Units, below). This allows you to use standard triggers in ME to handle successful scouting (especially in conjunction with the priority target list). These flags are defined in Recon Mode's config zone.

Name	Value	Description
prio+	Number	Increase this flag each time a unit that is listed as a priority target is detected. Default off
detect+	Number	Increase this flag each time an enemy unit is detected. If this unit is on the priority target list, this flag is not increased. Default off

## Priority Units

Recon mode supports a list of priority targets. They are detected normally, but their detection is handled differently: the detection event is different, and a different flag in ME is increased.

## Blacklisted Units

Recon mode also supports the exact opposite of priority items: blacklisted units. These are units that a recon plane never detects. Use it to hide strategic units from recon's prying eyes to ensure that they have to be discovered the old-fashioned way.

## Callbacks (Lua Only)

Guardian Angel supports callbacks so scripts can be informed whenever some recon events occur. Your callback must follow the following profile:

```
function demoReconCB(reason, theSide, theScout, theGroup, theName)
```

When invoked, `reason` contains the `reason` for the invocation, `theScout` is the unit that created the event, `theGroup` is the group that was spotted, and `theName` is the name of that group (in the case of a “dead” event the name of the scout unit that died. Currently, the following reasons are defined:

- “detected”  
A new group of ground troops was detected. If a priority group was found, this event is not invoked, but a “priority” event (see below) instead
- “removed”  
A mark that was placed on the map was removed. This is only invoked if a mark was placed (i.e. `applyMarks` is set to true)
- “priority”  
A group that is listed on the priority list was discovered. Note that for this event no “detected” event is invoked, just the “priority”
- “start”  
A scout has started their reconnaissance mission. `theGroup` is nil, and `theName` contains the string “<none>”
- “end”  
A scout has ended reconnaissance. `theGroup` is nil, and `theName` contains the string “<none>”.
- “dead”  
A scout has died while performing recon. Since the unit no longer exists, the **parameters contain unusual information**: `theSide` is -1, `theGroup` is nil, `theScout` is nil and **`theName` contains the name of the scout unit that died.**

Your script can sign up to be invoked with

```
cfxReconMode.addCallback(theCB)
```

### 3.5.10.2 Dependencies

Recon Mode requires `dcsCommon` and `cfxZones`

### 3.5.10.3 Module Configuration

Recon Mode can use a configuration zone for setting up options. To configure this module via a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it “reconModeConfig” (note: name must match exactly)

- Add any of the following attributes to this zone:

Name	Description
verbose	A value of “true” turns on debugging messages. Default is “false”
autoRecon	<p>If true, all planes are automatically treated as actively reconnoitering.</p> <p><b>NOTE</b> This is on by default. To avoid excessive scouting activity, you should reduce the number of active scout planes with enabling or disabling one of the following attributes: redScouts (off), blueScouts (off), greyScouts (off), playerOnlyRecon (on) Default: true</p>
redScouts	If true, all red planes are included as scouts when autoRecon is true. Default is false
blueScouts	If true, all blue planes are included as scouts when autoRecon is true. Default is <b>true</b>
greyScouts	If true, all neutral planes are included as scouts when autoRecon is true. Default is false
playerOnlyRecon	<p>If true, only player aircraft are included as scouts when autoRecon is true. All planes will not be automatically included as scouts.</p> <p><b>IMPORTANT</b> This condition is applied <b>in addition</b> to blueScouts and redScouts. If you disallow red scouts, red players will not automatically be added to the list of scouts. Defaults to false</p>
reportNumbers	If true, the F10 map markings include a unit count of the group at the time the group was discovered. Default is true
applyMarks	If true, discovered groups are marked on the F10 map. Default is true
announcer	If true, discovered groups are announced via text. Default is true
detectionMinRange	The detection range of a recon plane under worst conditions (low-level flying). Default is 3000 (3 km)
detectionMaxRange	The detection range of a recon plane under best conditions (high-altitude). Default is 12000 (12 km)
maxAlt	The altitude at which a plane achieves maxDetectionRange. Default is 9000 (9 km, 27'000 ft)
prio+	A flag in ME that is increased every time that a priority unit is detected
detect+	A flag in ME that is increased every time that a normal (non-priority) is detected
reconSound	The name of the sound file to play when a recon event occurs. Defaults to <nosound>, which will not play a sound

#### 3.5.10.4 ME Attributes

None.

### 3.5.10.5 API

Recom Mode provides a very simple API so mission designer can add and remove units to black list, prio list, and add and remove units to the list of scout/recon planes. For more advanced users, it also provides hooks for event callbacks

#### 3.5.10.5.1 addToPrioList(aGroup)

Adds aGroup to the list of groups that are priority targets. aGroup can be a string (group name) or the DCS group

#### 3.5.10.5.2 addToBlackList(aGroup)

Adds aGroup to the list of groups that will never be discovered by scouts. aGroup can be a string (group name) or the DCS group

#### 3.5.10.5.3 addScout(theUnit)

Adds theUnit as a scout/recon unit. theUnit can be a string (unit name) or DCS unit

#### 3.5.10.5.4 addCallback(theCB)

Adds theCB to the list of callbacks that are invoked on a recon event. theCB must match the following profile:

```
demoReconCB(reason, theSide, theScout, theGroup, theName)
```

with reason being a string, theScout a Unit, theGroup a group, and theName a string.

### 3.5.10.6 Using the module

Include the cfxReconMode source into a DOSCRIPT Action at the start of the mission

Optionally, add a config zone with ME

### 3.5.11 ssbClient

“[SSB](#)” is a freely available, multiplayer-only server module that allows aircraft ‘slot blocking’. SSB must be installed on the hosting server (and only the server). ssbClient is a mission (client-side) plug-in that allows mission designers to intelligently use slot-blocking ability of SSB in their missions. This means that by including ssbClient into your mission you can use slot-blocking functionality in your missions. Note that ssbClient only works in conjunction with SSB, and therefore requires the mission be run in multiplayer mode and an SSB-enabled server.

#### 3.5.11.1 Description

ssbClient provides automatic slot blocking for aircraft that

- have their starting location on an airfield/FARP that is currently occupied by the enemy (optionally neutral as well) – note that this is a dynamic feature, and when the airfield is captured, the aircraft become available (or blocked)
- are on an airfield that is “closed” ssbClient provides an API to open/close airfields
- are associated with an airfield that is occupied by the enemy (aircraft slots can be associated with the provided API)
- (optionally) aircraft that have crashed (a ‘single-use’ feature to prevent crashed aircraft to be re-used). This option provides a “re-use after” feature to allow access to the crashed aircraft slot after some time (to simulate replacement)

Note that a mission that enables the “single use” feature **requires that the host first disables** SSB’s automatic “**kickReset**” option.

Any player group that you wish to be blocked from spawning until the airfield belongs to the correct side must have the group's first player unit placed on the ground (i.e. "Take off" with one the following: "From Runway", "From Parking Area", "From Parking Area Hot", "From Ground Area", "From Ground Area Hot") within 3000m of the airfield's/FARP's center. That is all.

There are some additional (Lua-only) advanced options available (see below).

### Additional Features (Lua-Only)

The client supports methods to "close" and "open" airfields, and the ability to "bind" (and "unbind") aircraft to airfields. Using these methods is not required. They can be invoked either through your Lua scripts, or with a DOSCRIPT action in ME

#### *Opening and Closing an Airfield*

A closed airfield will not permit any player groups that start from there to be entered (slots are blocked), no matter who the airfield belongs to. This is commonly used to close FARPS that are still hidden, or to close contested airfields

You only need to open an airfield if it was previously closed, meaning that initially, all airfields are open.

#### *Binding/Unbinding a Group to an Airfield*

Due to the way that airfields are matched to groups, in rare cases it may become desirable to allow a player group to spawn in an enemy controlled airfield, FARP or even ship (or

ground-start close to it). Usually, SSBClient will prevent you from doing so, because it sees the closest airfield as enemy controlled. You can 'unbind' the group from any airfield, making it always available to start, no matter who owns the closest airfield.

Conversely, you can also bind groups to other airfields (i.e. different to those that SSB binds them to during start-up). Note that this does *not* move the aircraft to the newly bound airfield. It merely *binds the availability of that group to the ownership of the newly bound airfield*. Use this for special effects like allowing aircraft to become available on airfield B depending on the ownership of airfield A (by binding the group located at B to airfield A).

### ***Binding In-Air starts***

You can use above bind feature to bind in-air starts to airfields (usually, in-air starts are ignored by SSBClient). This requires that you ensure that `keepInAirGroups` is set to true (see module configuration, below).

#### ***3.5.11.2 Dependencies***

ssbClient requires dcsCommon, cfxGroups and cfxZones

#### ***3.5.11.3 Module Configuration***

ssbClient supports a convenient configuration zone to set it up for your mission's requirements.

To configure ssbClient via a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it "cfxSSBClientConfig" (note: name must match exactly)
- Add any of the following attributes to this zone:

Note that there are no mandatory attributes for ssbClient, so it can work out-of-the-box for most missions (and without requiring a config zone)

Name	Description
verbose	A value of "true" turns on debugging messages. Default is "false"
singleUse	A value of "true" turns on single-up: an airframe is blocked after crashing it. Note that this requires that the server's SSB setup sets kickReset to false in SSB. Defaults to false
reUseAfter	If singleUse is <b>enabled</b> , this optional attribute controls after how long a delay (in seconds) the slot may be re-used. This can simulate replacements arriving after some time. Setting this value to -1 blocks the slot for the remainder of the mission. Defaults to -1
allowNeutralFields	If set to "true", aircraft can spawn on neutral airfields (otherwise they are blocked). Defaults to false
maxAirfieldRange	Maximum range to find an airfield/FARP for a 'from ground' start. If no airfield is found that slot will be permanently open.
keepInAirGroups	For performance reasons, ssbClient strips all slots for air-starting aircraft from its observation list. In some cases (e.g. when you want to bind the availability of an air-starting aircraft slot to the ownership



Name	Description
	of an airfield) ssbClient must also manage air-starts. Set this value to true to also retain air-starting slots. Defaults to false
enabledFlagValue	This reflects SSB's flag value of that same name. <b>DO NOT CHANGE THIS UNLESS YOU ARE ABSOLUTELY SURE YOU KNOW WHAT YOU ARE DOING.</b> Defaults to 0
enabledFlagValue	This reflects SSB's flag value of that same name. <b>DO NOT CHANGE THIS UNLESS YOU ARE ABSOLUTELY SURE YOU KNOW WHAT YOU ARE DOING.</b> Defaults to enabledFlagValue + 100

#### 3.5.11.4 ME Attributes

None.

#### 3.5.11.5 API

##### 3.5.11.5.1 `closeAirfieldNamed(name)`

A closed airfield will not permit any player groups that start from there to be entered (slots are blocked), no matter who the airfield belongs to. This is commonly used to close FARPS that are still hidden, or to close contested airfields. Name is the name of the FARP or airfield, and must match exactly

##### 3.5.11.5.2 `openAirFieldNamed(name)`

You only need to open an airfield if it was previously closed, meaning that initially, all airfields are open. Name is the name of the FARP or airfield and must match exactly

##### 3.5.11.5.3 `unbindGroup(groupName)`

Due to the way that ssbClient automatically matches FARPs/airfields to groups, it may become desirable to allow a player group to spawn in an enemy controlled airfield, FARP or even ship (or ground-start close to it). Usually, SSBClient prevents you from doing so, because it sees the closest airfield as enemy controlled. You can 'unbind' the group from any airfield, *making it always available to start*, no matter who owns the closest airfield (single-use restrictions may still apply, though).

groupName is the name of the group that is to be unbound

##### 3.5.11.5.4 `bindGroupToAirfield(groupName, airfieldName)`

You can bind groups to other airfields (i.e. different to those that SSB binds them to during start-up). Note that this does not move the aircraft to the newly bound airfield. It merely binds the availability of that group to the ownership of the newly bound airfield. Use this for special effects like allowing aircraft to become available on airfield B depending on the ownership of airfield A (by binding the group located at B to airfield A).

groupName is the name of the group, airfieldName that of the new airfield to bind slot accessibility to. Both must match exactly.

#### 3.5.11.5.5 `openSlotForCrashedGroupNamed(gName)`

You can manually ‘force’ re-opening of a group after a crash by invoking this method. `gName` must match exactly the name of the group. If the group wasn’t blocked, this call is ignored.

#### 3.5.11.6 *Using the module*

Include the `cfxSSBClient` source into a DOSCRIPT Action at the start of the mission

To change any configuration settings, add a `SSBClientConfig` zone with the relevant attributes.

### **3.5.12 ssbSingleUse (tbc)**

*3.5.12.1 Description*

*3.5.12.2 Dependencies*

*3.5.12.3 Module Configuration*

*3.5.12.4 ME Attributes*

*3.5.12.5 API*

*3.5.12.6 Using the module*

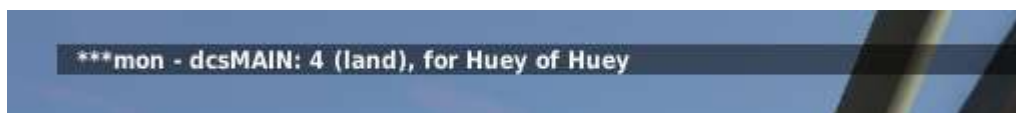
Remember to turn off kickReset

### 3.5.13 cfxMon Development Tool (Lua Only)

#### 3.5.13.1 Description

cfxMon is an event monitor for mission designers who are developing Lua-based scripts and want to visualize/analyze which events and in what order events are generated during a mission. cfxMon has built-in support for most DML modules and automatically installs its own callbacks for the modules it finds on start-up.

Each time an event is generated, it is logged to the screen, and context added when available. For example, if DCS generates the world event 4, this is what cfxMon could put to the screen:



In above example, the event ID is 4, and cfxMon translates that to “land” automatically. It was invoked by unit “Huey” of the group “Huey”.

You can selectively turn logging on and off for any DML module through a config zone.

#### 3.5.13.2 Dependencies

cfxMon requires dcsCommon and cfxZones.

Optional: all other DML modules

#### 3.5.13.3 Module Configuration

cfxMon uses a configuration zone that allows you to enable/disable specific event monitoring. To use a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it “monConfig” (note: name must match exactly)
- Add any of the following attributes to this zone. Adding “no” or “false” turns off that monitoring ability. All values default to true:

Name	Description
dcsCommon	Log all DCS world events
cfxPlayer	Log DML Player events. Only monitors these events if cfxPlayer module is present, disabled otherwise
cfxGroundTroops	Log DML Ground Troops events. Only monitors these events if cfxGroundTroops module is present, disabled otherwise
cfxObjectDestructDetector	Log DML object destruct events. Only monitors these events if cfxObjectDestructDetector module is present, disabled otherwise
cfxSpawnZones	Log DML spawn zone events. Only monitors these events if cfxSpawnZones module is present, disabled otherwise
(others)	(to follow)
delay	Time (in seconds) that a log is displayed on the screen. Defaults to 30. Set it to a shorter time (and use above

Name	Description
	attributes to turn off events) when you are getting flooded with events and the screen can't keep up with showing them.

#### 3.5.13.4 ME Attributes

None.

#### 3.5.13.5 API

None.

#### 3.5.13.6 Using the module

Include the cfxMon source into a DOSCRIPT Action at the start of the mission

To change any configuration settings, add a monConfig zone with the relevant attributes.

### **3.5.14 Module Name**

*3.5.14.1 Description*

*3.5.14.2 Dependencies*

*3.5.14.3 Module Configuration*

*3.5.14.4 ME Attributes*

*3.5.14.5 API*

*3.5.14.6 Using the module*

## 3.6 Using Foundation (Lua Only)

### 3.6.1 dcsCommon (Lua only)

#### 3.6.1.1 *Description / Using dcsCommon*

`dcsCommon` is the ‘bedrock’ module. All other modules require its presence. This module provides:

- A replacement for DCS’s world event notifier that is more modular and flexible
- A library of often-used methods to more easily perform common tasks (i.e. a library that any designer would have to write anyway and for which there is little excuse for ED not to have provided)

Including it into a mission does not induce any performance penalties, as its methods are all passive, so it can be added to any mission, including those that utilize other libraries such as MIST or MOOSE.

`dcsCommon` provides a collection of methods that are described in the API

#### Event Handling

`dcsCommon` provides a convenient, more flexible event handler for mission designers. Instead of simply calling your code when something happens, invocation is done in multiple, conditional stages. You can optionally provide a callback for each stage and thus have a more fine-grained control over what happens, and separate out code blocks

When you subscribe to `dcsCommon`’s event handler, you can pass up to four different callback methods:

```
addEventHandler(f, pre, post, rejected)
```

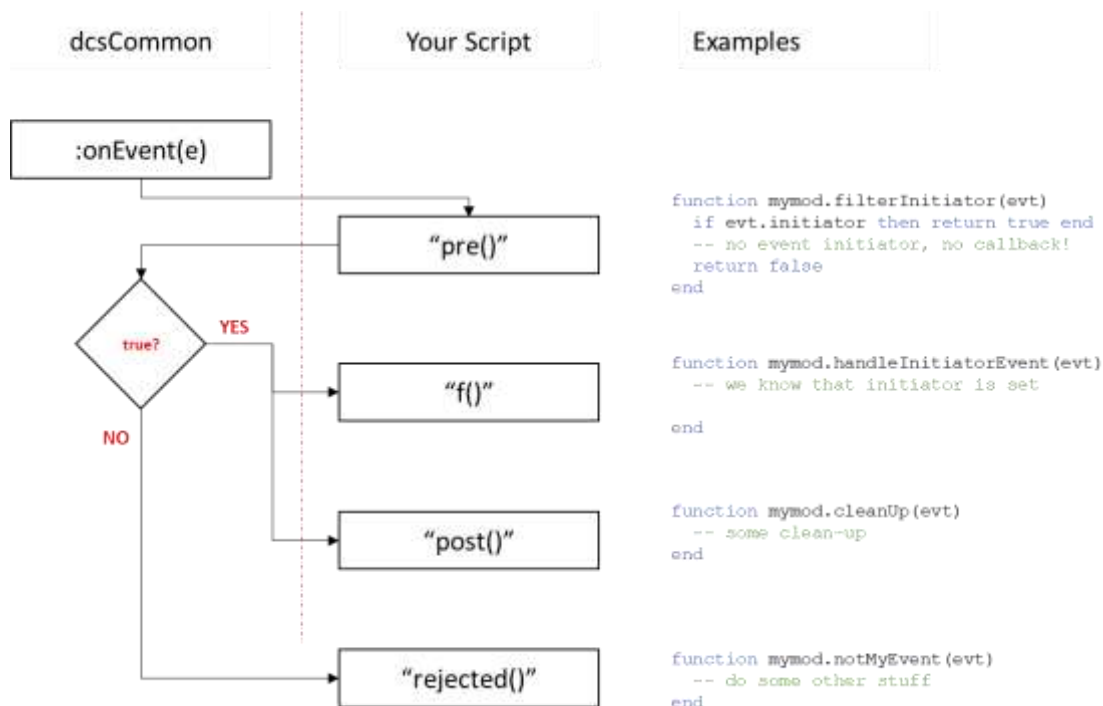
with

- `f` being the actual event handler that is to be invoked when `pre` returns true
- `pre` is the optional pre-processor that determines if `f` should be invoked. Return true to have `f` invoked.
- `post` is the optional post-processor that is only invoked if `f` was invoked
- `rejected` is the optional reject-processor if `pre` returned false

All callbacks have the same prototype

```
function mymodule.someName(event)
```

and `event` containing the event table as passed from DCS.



dcsCommon provides another, streamlined version that allows the author to pre-select the events they want to be invoked for:

```
addEventHandlerForEventTypes(f, evTypes, post, rejected)
```

which essentially replaces `pre` with an array of numbers (the event IDs) in `evTypes` that list the IDs that you want `f` to be invoked for:

As with `addEventHandler`,

- `post` is an optional post processor invoked after `f`
- `rejected` is an optional method that is invoked when `evTypes` does not contain the current event ID

## String / Table Handling

dcsCommon implements a number of convenient methods to enumerate tables (convert them to arrays), choosing random elements and similar.

Also present are often-used methods for string manipulation, like splitting strings into arrays (for example a string containing a comma separated list of units into an array), trimming strings (removing leading and trailing blanks), testing if a string starts with or ends with a string, and remove that string etc.

## Vector math

Don't be afraid of that name – it's merely a collection of methods that makes it easy to deal with locations (points) and directions (both are called vectors in mathematics). These methods allow you to determine locations, move one point gradually towards another etc.



## Miscellaneous

The grab bag of function provides convenience access to mission designers that don't fit a particular niche, but are still common fodder: getting the bearing to a point, calculating closing distance, getting the altitude of a unit, measuring the distance between two points etc. All here for convenience.

## Group/Unit handling

These are convenience methods to quickly access units in groups, find groups close to points, and get information about individual units like their heading, speed etc.

## Spawning Groups

Spawning units in DCS is a process that involves multiple steps:

- create a group data table. This is the container for all units. A group always contains units of the same category: ground, ship, aircraft, helicopters. Air units have to always be the same type (e.g. all have to be A-10A or all have to be UH-1), while ground and water groups can mix types (e.g. Hummer and Leopard II), but all have to be in the same category (ground). Paths / Routes and orders are always given on the group level). All groups must be named, and a group name must be unique across all other groups in the mission.
- Provide the "type string" information that tells the game which object model to use
- create path and order data tables for the group, and add them to the group
- create unit data tables for each unit of a group and add them to the group. Unit-individual properties are location, heading and name. Names must be unique across all units in the mission.
- air units are similar to ground units and may require you to add additional data tables like loadout and callsigns
- once the group's data table is complete, spawn the group in DCS. It is only at this point where you specify which coalition the units belong to, which is determined by the country code.

`dcsCommon` provides a number of convenience methods that can streamline this process and even arrange the groups in 'formations'. To simply spawn a single unit with a type string of "Soldier M4" at a certain location, with a set heading and no orders, you can use the method `createSingleUnitGroup()`. That method will create the group and unit data tables for you. You then invoke `coalition.addGroup()` with that group data to spawn the unit/group on the map:

```
-- create group data
local theGroupData = dcsCommon.createSingleUnitGroup("test123",
"Soldier M4", x, z, 270)
-- spawn in world
local theGroup = coalition.addGroup(1, groundCat, theGroupData )
```

A more advanced sibling `createGroundGroupWithUnits()` can arrange a number of units (passed as an type string array) into a formation:

```
-- create group data from multiple unit type strings
local theGroupData = createGroundGroupWithUnits("abc4", {"Soldier
M4", "Soldier M4", "Soldier M4"}, 20, 3, "circle_out", 0)
-- spawn in world
local theGroup = coalition.addGroup(1, groundCat, theGroupData )
```

### **Spawning Static Objects, linked objects**

Spawning static objects follows similar rules, except that you can omit the group and spawn directly.

dcsCommon supports spawning linked objects, i.e. objects that move with ships. Due to the complexity involved with offsets and rotation, it is strongly recommended to place linked objects in ME using objectSpawnZones instead of directly invoking methods from dcsCommon, as it takes significantly less effort, works, and supports ME's GUI.

### **Airbase / FARP / Ship handling**

dcsCommon provides mission designers with several convenience methods to find and filter airbases based on

- parts of their name (e.g. 'senaki' will return Senaki Kolkhi)
- their type (aerodrome, FARP or ship)
- their coalition
- closest distance to points or units

There are also convenience accessors to get parking slots on airfields. There are also methods to create waypoints involving airfields for aircraft units (take-off, overhead, landing)

#### **3.6.1.2 API**

dcsCommon's entire (large) API is described in its own chapter → dcsCommon API

#### **3.6.1.3 Dependencies**

None

#### **3.6.1.4 Module Configuration**

None

#### **3.6.1.5 ME Attributes**

dcsCommon has no ME interface

### 3.6.2 cfxPlayer (Lua Only)

cfxPlayer is a collection of methods that mainly simplify tasks revolving **around player-controlled units and groups with player-controlled units**.

#### IMPORTANT

cfxPlayer differentiates between players and the units these players occupy, and they are separate entities with different life cycles. Units/groups can appear multiple times over the course of a mission, while players can only appear, and will stay until the end of the mission. Player „A“ always is player „A“, no matter which side or unit they control.

Similarly, cfxPlayer does not care who controls a player unit: if networked player “netA” first controls unit “myUnit”, then changes to a different unit, and later a networked player “netB” controls “myUnit”, to cfxPlayer they both are “myUnit”, no matter who controls it - even if that unit respawned multiple times.

cfxPlayer is useful for mission designers who create scripts that offer functionality based on individual players (e.g. score keeping, UI). Note also that many of DML’s higher-level modules require cfxPlayer.

#### 3.6.2.1 Description / Using cfxPlayer

Mission designers rarely invoke cfxPlayer methods directly. Instead, their main entry points are

- **Player event callback**

Whenever something happens that changes the context of a player-controlled unit or group from a game’s perspective, cfxPlayer invokes callbacks with a description of the event. Mission designers subscribe to these events by providing a simple callback to

```
cfxPlayer.addMonitor(callback, events)
```

- **Current player info DB**

cfxPlayer maintains an up-to-date DB of all current player-controlled units and groups that contain player-controlled units. Scripts can access this DB for their own requirements

- **Current player group DB**

cfxPlayer maintains an up-to-date dictionary of all current groups that contain player-controlled units

- **Current player DB**

cfxPlayer maintains an up-to-date dictionary of all currently connected players and the names of the unit they occupy (or “<none>” if they multicrew in a slot different from pilot)

- **(occasionally) Convenience accessors to player information**

Occasionally, you may want to access information about a player-controlled unit, like their current airframe, the first player-controlled unit in a group, or the player-

controlled unit's position etc. `cfxPlayer` provides a number of convenience accessors for this.

`cfxPlayer` regularly checks the status of all players, player-controlled units, and groups that contain player units, tracks their current status, and invokes callbacks whenever it detects a change.

### The Player Unit/Group DBs

`cfxPlayer` provides globals (and accessor) that contain up-to-the-second current information about all player-controlled units, and groups with player-controlled units in the game. When you access these DBs it is important that you **do not make any changes** to the data, or `cfxPlayer` may return unpredictable results

The following DBs are available for mission designers:

- `cfxPlayer.playerDB`  
somewhat misleadingly named, this contains information of all currently player-controlled **units**
- `cfxPlayerGroups`  
this **global** is a DB that tracks all groups that contain at least one player-controlled unit
- `cfxPlayer.netPlayers`  
a dictionary with player names as keys, and the unit names of the units they occupy as values. **WARNING:** multi-crew units only show the pilot, so any **crew members do not show** up as players! Updated once a second

Each entry to the `playerDB` is a table with the following attributes:

- `name`  
Name of the **player's unit**. THIS IS NOT THE PLAYER'S NAME
- `unit`  
the unit that the player controls
- `unitName`  
the name of the unit that the player controls. Same as `name`
- `group`  
the group that the player's unit belongs to
- `groupName`  
the name of the group that the player's unit belongs to
- `coalition`  
the coalition that the player's unit is aligned with

## Player Event Callbacks

When a script subscribes to cfxPlayer's event notifier, it passes a method that matches the following profile:

```
myPlayerEventCallback(eventType, description, info, data)
```

with

- `eventType` being a string containing the event category, e.g. 'new'.
- `description` being a human-readable string describing the event that can be easily dumped to log or displayed on-screen
- `info` being the relevant record table from the player unit DB
- `data` being a table containing additional information, dependent on the event

There is an important difference between how cfxPlayer *player* events versus *unit/group* events are generated: players can only appear as new once, and won't disappear over the course of the mission. Units and groups can appear and disappear as they are occupied by player. This means, that there is exactly one `newPlayer` event for each player, and no event for players leaving. Even if a player disconnects from the server, cfxPlayer keeps their record until the end of the mission. Conversely, units and groups can appear and disappear for the same and other players multiple times while the mission is running.

## cfxPlayer Status Change Event Types

cfxPlayer invokes any subscribed callbacks with the following event types

- **"new"**  
Player's *unit* did not exist in DB before and now has appeared for the first time. Note that player units spawn only when they are inhabited.  
Note that cfxPlayer tracks player units, not the player who controls them

`playerInfo` contains a record of the player's unit  
`data` is an empty table

- **"side"**  
The player's unit has changed sides (e.g. RED to BLUE)

### NOTE:

This can't happen, because currently units can't change sides. This event is provided in case this possibility appears in DCS

`playerInfo` contains a record of the player's unit  
`data` contains the two attributes `old` and `new` with the old and new coalition values, respectively

- **"group"**  
Player's unit has changed to a different group. This event has the same caveats as 'side', as usually, a unit can't change groups, and cfxPlayer tracks player units, not the player who controls the unit. Present in case DCS introduces that ability for units

`playerInfo` contains a record of the player's unit  
`data` contains the two attributes `old` and `new` with the old and new coalition values, respectively

- **"unit"**  
unit is inhabited by a different player. This event currently isn't detected, since `cfxPlayer` tracks player units, not individual players. It's present for later extensions.  
`playerInfo` contains a record of the player's unit  
`data` contains the two attributes `oldUnitName` and `new` with the old unit's name (not unit, as it may no longer exist) and new unit, respectively

- **"leave"**  
A formerly player-controlled unit has disappeared from the game  
`playerInfo` contains a record of the player's unit. Note that while the unit itself is not longer valid, the unit's name is still retained correctly  
`data` contains an empty table

- **"newGroup"**  
A new group containing a player-controlled unit has appeared.  
`playerInfo` is nil  
`data` contains a table with the attributes: `group` (the group that appeared), `name` (the group's name), `primeUnit` (the first unit in that group that is player-controlled), `primeUnitName` (the name of that unit), and `id` (the group's ID)

- **"removeGroup"**  
A group that formerly contained player-controlled units either no longer contains any player-controlled crafts any more, or has disappeared entirely  
`playerInfo` is nil  
`data` contains a table with the attributes: `group` (the group that appeared), `name` (the group's name), `primeUnit` (the first unit in that group that is player-controlled), `primeUnitName` (the name of that unit), and `id` (the group's ID)  
`data` contains the group that no longer contains player-controlled units. This group may no longer be valid

- **"newPlayer"**  
A new player has appeared. Since player detection is bound to them occupying the pilot slot of any plane, including multicrew, new players are detected only when they enter a pilot slot in a unit. Once detected, a player will remain in the DB until the end of the mission, even if they disconnect  
`playerInfo` is nil  
`data` contains a table with the attributes: `playerName` (the name of the player that appeared), `newUnitName` (the name of the unit the player is controlling as pilot)

- **"changePlayer"**

A player has changed to a different unit, or left the unit they formerly occupied.

`playerInfo` is nil

`data` contains a table with the attributes: `playerName` (the name of the player that has changed units), `newUnitName` (the name of the unit the player is controlling as pilot, or "<none>" if no unit), `oldUnitName` (the name of the unit the player has left, or "<none>" if they didn't control a unit before)

**Note:**

"changePlayer" can be **invoked twice**: once for leaving the old plane (without entering a new one), and once for entering the new plane. In the former case (player left unit), `newUnitName` is '<none>'.

From above, the events that currently are relevant for mission designers are

- for units: `new` and `leave`
- for groups: `newGroup` and `removeGroup`
- for players: `newPlayer` and `changePlayer`

### Example Sequence Of Events

To better illustrate how `cfxPlayer` works, let's look at what may happen in a game, and what events will be invoked coded by color for `group`, `unit` and `player`:

1. **Game starts**

- No event

2. Player A **chooses side RED**

- No event

3. Player A **occupies** "Frogger A" that is part of group "Redfrogs"

- "new" event for unit "Frogger A"
- "newGroup" event for group "Redfrogs"
- "newPlayer" event for "A"

4. Player A **changes** to unit "Fulcrum B", part of "Redcalf" group

- "new" event for unit "Fulcrum B"
- "leave" event for unit "Frogger A"
- "newGroup" event for group "Redcalf"
- "removeGroup" event for group "Redfrogs"
- "changePlayer" event for player A

5. Player A **changes** to BLUE **side** while occupying RED unit

- "leave" event for unit "Fulcrum A"
- "removeGroup" event for group "Redcalf"

6. Player A chooses unit "Hogger" in group "Winged Pigs"

- "new" event for unit "Hogger"

- "newGroup" event for group "Winged Pigs"
  - "changePlayer" event for player A
7. Player **crashes** "Hogger"
    - "leave" event for "Hogger"
    - "removeGroup" event for "Winged Pigs"
  8. Player enters unit "Bug's Bunny" of group "Hornets"
    - "new" event for unit "Bug's Bunny"
    - "newGroup" event for group "Hornets"
    - "changePlayer" event for player A
  9. Player **ejects** from unit "Bug's Bunny"
    - "removeGroup" event for group "Hornets"
  10. Now **derelict** plane "Bug's Bunny" **crashes**
    - "leave" event for unit "Bug's Bunny" (yeah, strange)
  11. Player A enters unit "A new Hopper" in group "Hueys"
    - "new" event for unit "A new Hopper"
    - "newGroup" event for group "Hueys"
    - "changePlayer" event for player A

### 3.6.2.2 API

The entire API for cfxPlayer is described in → cfxPlayer API

### 3.6.2.3 Dependencies

Required Modules

- dcsCommon must be loaded

### 3.6.2.4 Module Configuration

- cfxPlayer.verbose – when true, events are printed to screen when they happen

### 3.6.2.5 ME Attributes

cfxPlayer does not use ME Attributes

### 3.6.2.6 Using the module

Include cfxPlayer source into a DOSCRIPT Action at the start of the mission



### 3.6.3 cfxZones (Lua Only)

cfxZones is one of the most fundamental modules that DML is built upon, and it provides most of the heavy-lifting for DML's ME integration. This module provides a replacement for DCS's built-in Trigger Zones and allows mission designers to enhance and expand on them (if they use DML's zones instead of those provided by DCS).

#### IMPORTANT

Although most of the information contained in this section is of technical nature and accessible only via Lua, I recommend that you at least read the part about **Zone Attributes** and **Zone Names**, as they are used extensively from within ME

#### 3.6.3.1 Description

cfxZones both provides the essential infrastructure for managing zones, and provides a wealth of zone-based methods for creating, modifying, testing and managing zones.

At the start of a mission, cfxZones reads all zones that designers previously set up in ME, and provides its own wrapper. Mission designers should always use these zone wrappers instead of using ME's Trigger Zones directly.

#### Zone Attributes

cfxZones provides **strong support for ME's zone attributes** (sometimes also called 'properties'), that designers can add, modify, and remove from inside ME. Zone attributes are text-based key-value pairs, and cfxZones allows direct access to these properties, as well as capabilities to search and collect zones based on attributes.

Many modules further up in the library utilize these abilities to implement their function simply by looking for the presence of an attribute and then control their functionality by from additional information taken from other attributes.

This simple mechanism allows designers to combine ("stack") functionality of multiple modules into a single ME Zone: the example on the right contains different properties that are used by different modules:

- the "**CSARBASE**" attribute indicates to the *CSARManager* module that the area of this zone is a legal area for helicopters performing a CSAR mission to return rescued personnel to
- the "**FARP**" attribute indicates to *FARPZones* that the zone is a conquerable FARP (with additional properties defining how that FARP defends itself for each side)



- the “**pilotsafe**” attribute signals to the *limitedAirframes* module that air frames that are landed inside this zone can be safely exited (the player can change to a different airframe/slot) without losing a pilot (provided, of course, that the zone is owned by the correct side).

Please read the relevant sections of this documentation to learn more about those modules and which zone properties they use

We recommend you adopt similar schemes for your own extensions, as zone properties are easy to maintain in ME.

## Zone Names

Of course, *cfxZones* also supports access to zones by their name, or part of their name. Again, some modules in the library utilize zone names for deriving functionality. Be advised that using a name to derive functionality is an inferior method compared to properties. When it is used in DML modules, it is usually to identify fixed names, and they are used to store configuration information for the module, rather than mission-relevant information.

### NOTE:

Please be advised, that *cfxZones* treats zone names case **IN**sensitive! If you have two or more zones in your mission that differ in name only by their upper/lower case spelling, you will lose access to all of them except one. Ensure that you give unique names to all your zones in ME.

## Standard Zone Attributes

All *cfxZones* have at least the following properties:

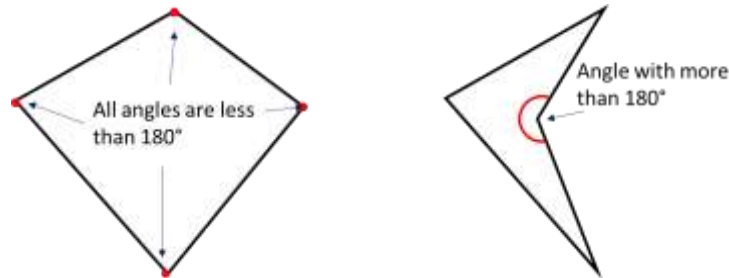
- **point**  
A vector (x, 0, z) that represents the zone’s ‘main’ (central) point on the map. Y (height) is always zero.  
**Never access this property directly!** *cfxZone*’s main accessor for a *cfxZones*’s point is *cfxZones.getPoint()* which will return the zone’s current location, updated in case the zone is a moving zone. The point returned is a copy of the zone’s main point, altering it will not change the zone’s position.
- **isPoly**  
A Boolean that is false if the zone is set up as a circle in ME, true otherwise.
- **isCircle**  
A Boolean that is true if the zone is set up as a circle in ME, false otherwise.
- **radius**  
The radius of the zone, as defined in ME. Note that poly zones also have a radius. Strange, but they do.

## Circular and Quad Zones: compatibility and restrictions

*cfxZone*’s point testing methods (e.g. *isPointInsideZone()*) are compatible with ME’s

Quad zones. However, note that currently, quad zone testing may fail for some quad zone shapes. `cfxZones` uses a fast point inside polygon testing algorithm that assumes convex polygons (which most quad zones are, since a polygon in ME currently is limited to four corners and no crossing lines).

A convex shape is a shape where all angles seen from inside the shape must have an angle of 180 degrees or less:



If your quad zone has a non-convex (concave) shape (as pictured above on the right), some inside testing can fail for that zone.

### Moving Zones / Linked Zones

`cfxZones` supports simple methods to link zones to the movement of units. If you add a `linkedUnit` property to a unit in ME, the value of that property determines (by name) the unit that this zone is linked to. Note that the unit to link to must exist and the name must match exactly, or the zone is immediately unlinked and won't link even if a unit of that name is later created.

There are multiple ways that a zone follows a unit:

- *Centered*  
The zone's center is placed on the unit's position (default). This way the zone's center always moves with the unit. Even if you initially create the zone some distance away from the unit to follow, the zone will always be placed directly above the linked unit.
- *Offset*  
The zone always keeps the same displacement relative to the unit, as set up in ME during mission creation. If the mission starts with such a moving zone places 300 meters southeast of the linked unit, `cfxZones` always updates the zone to remain 300 meters southeast of the unit even if the unit moves to a different location.  
To make a zone keep its offset relative to the unit it is linked to, use ME to add the property `useOffset` with a value of `yes` or `true` to the zone  
Note that the linked unit's orientation does not influence the placement of the zone. If the linked unit turns by some 180 degrees, the zone will still be placed 300 meters southeast of the unit (and not, as some might have expected, 300 meters to the northwest: the unit moves relative to the unit's location, not the units heading).

Note also that you can create zones and link them via script using the API. This means that you can use scripts to late-link (i.e. after the mission starts) units and zones that did not exist at the beginning of the mission.

### 3.6.3.2 *Dependencies*

cfxZones requires dcsCommon to be loaded

### 3.6.3.3 *Module Configuration*

cfxZones does not require to be configured

### 3.6.3.4 *API*

Please see the dedicated chapter → cfxZones API for more information

### 3.6.3.5 *ME Attributes*

### 3.6.3.6 *Using the module*

Include the cfxZones source into a DOSCRIPT Action at the start of the mission

### 3.6.4 cfxCommander (Lua Only)

cfxCommander is mainly a middleware module that mission designers use to facilitate issuing orders to ground troops. In missions, orders are usually given in a sequence, and one of cfxCommander's main features (besides simplifying order construction) is the ability to schedule issuing orders, so mission designers can give a string of orders: "stop all vehicles now, then attack this enemy in 10 seconds"

Another central ability of cfxCommander revolves around pathing: this module interacts with 'pathing' zones that mission designers can place on the map. These zones give hints to cfxCommander how to utilize roads.

cfxCommander is one of the central modules for cfxGroundTroops

#### 3.6.4.1 Description

cfxCommander is mainly used by other modules, and by mission designers that require to issue move orders to ground units via scripts utilizing cfxCommander's API.

#### Configuration Zone

cfxCommander's main options can be configured using a simple Trigger Zone. See → Module Configuration, below

#### Pathing via roads

cfxCommander has built-in abilities to create path orders for groups from one point on the map to another, and can use or disregard roads based on the modules configuration, specially placed 'pathing zones' and the API:

- *API*  
When requesting a path for a group's move order via API, the mission designer can specify if roads should be followed
- *Pathing Zones*  
Mission designers can place Trigger Zones in ME with a 'pathing' attribute that locally overrides any pathing as requested from the API. This allows mission designers to locally turn off road pathing when the area's road network is too complex
- *Configuration*  
Mission designers can place a special zone using ME that sets up global pathing (i.e. overrides any pathing zones and API).

#### Issuing Orders in cfxCommander

Ordering troops in DCS is a somewhat involved process, and cfxCommander can only simplify the process somewhat. DCS itself differentiates between "Tasks", "Commands" and "Options".

cfxCommander somewhat unifies this, offering an API that abstracts this to "orders" that are to be carried out by the group. The underlying code then resolves this to tasks, commands and options – whatever may apply. Additionally, cfxCommander allows you to schedule the execution of your orders. This is particularly important for two reasons:

- Ordering troops to do something immediately after they have been spawned may crash DCS, so you need to schedule your first orders a few seconds in the future
- Orders usually come as a sequence: first do this, then do that. cfxCommander recognizes this requirement and makes it easy to issue a sequence of orders that then are scheduled in the future.

cfxCommander has an API that allows mission designers to build and schedule their own tasks, options, or commands. Using that API withing mission code is discouraged; it's far better to emulate cfxCommander to build read-made 'orders' as a method, and then use that to issue orders. The 'orders' that cfxCommander currently support is limited, but usually sufficient for most missions

- Order a group to move to a location
- Order a group to stop
- Have a group transmit on the radio (once or continuous)
- Have a group stop transmitting

### **Formations**

when moving, ground troops in DCS can assume some pre-determined formations. These are distinct from formations they can be assembled in during spawning (see some modules later). Since cfxCommander places heavy emphasis on correct pathing, and DCS uses a special case of formations to control road pathing, formations are only available with some of the API, and may even be overridden then.

DCS has defined the following formations for ground units

- Off Road
- On Road
- Cone
- Rank
- Diamond
- Vee
- EchelonR
- EchelonL

#### *3.6.4.2 Dependencies*

cfxCommander requires dcsCommon and cfxZones

#### *3.6.4.3 Module Configuration*

cfxCommander can use a configuration zone for setting up main options. To configure cfxCommander via a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it "CommanderConfig" (note: name must match exactly)
- Add any of the following attributes to this zone:

Name	Description
verbose	A value of "true" turns on debugging messages. Default is "false"
forceOffRoad	if set to "true", vehicles path will generally follow roads, but may drive offroad (they follow a list of vertex points generated from the road, but often do not drive in the road itself). Default is "false"
noRoadsAtAll	Completely turns off road following when set to "true". Ground units path directly to the destination, using as few straight lines as possible. If set to true, this overrides any setting for "forceOffRoad". Default is "false"

#### 3.6.4.4 API

##### Low-Level

The low-level API methods are for assembling and setting DCS-Level order fragments, like *Options*, *Tasks* and *Commands*

##### Options

Options are set as key, value pairs as described in the DCS API

##### 3.6.4.4.1 `scheduleOptionForGroup(group, key, value, delay)`

Sets a group's key, value pair as described in delay seconds. Default for delay is 0.1 seconds

##### Commands

Commands in DCS terms are immediate actions like setting frequency or starting/stopping a transmission. From a coding standpoint, commands set a group's state. Commands can be assembled into a block, and then issued as a whole. The process of issuing commands is a follows:

- create a command data table that holds all commands to issue for a group
- create command(s)
- add command(s) to the command data table
- schedule execution of all commands in the data table

##### 3.6.4.4.2 `createSetFrequencyCommand(freq, modulator)`

returns a command for setting the frequency

##### 3.6.4.4.3 `createTransmissionCommand(filename, oneShot)`

returns a command for sending a sound file (as indicated by filename) via the currently set frequency. If oneShot is true, the mission ends after the sound file has played once in full, otherwise it repeats infinitely (or until the group is deleted or a stop transmission command is issued)

##### Note:

Omit the path that leads to the mission's main ("I10n/DEFAULT/") for the filename

#### 3.6.4.4.4 `createStopTransmissionCommand()`

returns a command to stop a transmission from this group.

#### 3.6.4.4.5 `createCommandDataTableFor(group, name)`

returns an empty command data table for group. The table's name is set to name. After adding commands to the table, pass it to the scheduling method for execution.

#### 3.6.4.4.6 `addCommand(theCD, theCommand)`

adds theCommand to the command data table theCD.

#### 3.6.4.4.7 `scheduleCommands(data, delay)`

schedules all commands contained in the command data table data to be executed after delay seconds.

#### 3.6.4.4.8 `scheduleSingleCommand(group, command, delay)`

A one-step shortcut that allows a single command to be scheduled for group after delay seconds without first having to wrap it in a command data table.

### Tasks

Tasks in DCS are what groups are doing. Groups usually either move, hold or attack. Since `cfxCommander` provides strong support for moving troops via "Orders", we only expose the API for task scheduling and creating Attacking and Engaging other groups.

#### 3.6.4.4.9 `scheduleTaskForGroup(group, task, delay)`

scheduled task to be assigned to group after delay seconds.

#### 3.6.4.4.10 `createAttackGroupCommand(theGroupToAttack)`

returns a task to attack theGroupToAttack that can be scheduled

#### 3.6.4.4.11 `createEngageGroupCommand(theGroupToAttack)`

returns a task to engage theGroupToAttack that can be scheduled. Readers discretion is advised to find out the difference between attacking and engaging groups.

### High-Level: Orders

Orders are an `cfxCommander` abstraction layer for easier access. They all "make a group do something". This may result in tasks, options or commands to be issued under the hood, but from the API level, they are all the same, and all can be scheduled.



**Important Note:**

Modules further up in the hierarchy (e.g. `cfxGroundTroops`) may also use their own definition of “orders”. They are different and distinct from another and can’t be mixed.

#### 3.6.4.4.12 `makeGroupGoThere(group, there, speed, formation, delay)`

Causes group to start moving to there at the indicated speed in delay seconds. On their way, they’ll assume formation. Formation must be one from the set defined by DCS (see above), e.g. “Cone”. If you want the group to follow roads, use `makeGroupGoTherePreferringRoads()` instead as that method provides automatic support for pathing zones and configuration preferences.

#### 3.6.4.4.13 `makeGroupGoTherePreferringRoads(group, there, speed, delay)`

This is the prime moving order for `cfxCommander`, and will path group from where they are to there at the indicated speed, starting in delay seconds. Pathing will observe pathing zones and config settings.

#### 3.6.4.4.14 `makeGroupHalt(group, delay)`

Orders group to stop after delay seconds

#### 3.6.4.4.15 `makeGroupTransmit(group, tenKHz, filename, oneShot, delay)`

Starts a transmission on the radio at  $\text{tenKHz} * 10000$  Hz (e.g. if `tenKHz` is 123 the frequency is set to 1230000 Hz = 1.23MHz), playing the sound file `filename` after delay seconds. If `oneShot` is set to true, the transmission ends after one play-through, otherwise it loops until the group is deleted, or a stop transmission order is issued

##### 3.6.4.4.15.1 `makeGroupStopTransmitting(group, delay)`

Stops group to transmit on the radio after delay seconds.

#### 3.6.4.5 *ME Attributes*

As described above, `makeGroupGoTherePreferringRoads()` observes pathing zones that mission designers can place with ME. Add the following attributes to control a group’s pathing that is calculated with that method. In order to influence a group’s pathing, at least start or end point of the path must be inside a pathing group.

Name	Description
pathing	Marks this ME Zone as a pathing zone. The value of this attribute gives <code>cfxCommander</code> hints on how to create paths that lead into, reside within, or lead out of this zone. Currently, the following hints are supported: <ul style="list-style-type: none"><li>• “normal” Create a path as requested</li><li>• “offroad” Orders to move on roads are ignored, and off-road pathing is</li></ul>

Name	Description
	<p>allowed. Use this in case an area has a complex road system that makes pathing difficult or degrades performance.</p> <p><b>MANDATORY</b></p>

#### 3.6.4.6 *Using the module*

Include the `cfxCommander` source into a DOSCRIPT Action at the start of the mission

To change any configuration settings, add a `CommanderConfig` zone with the relevant attributes.

### 3.6.5 nameStats (Lua Only)

Many missions require one way or another to track some important figures: be they score, weight or other meaningful statistics. nameStats is DML's number-tracking foundation. It can only be accessed via API

#### 3.6.5.1 Description

nameStats provides an easy to use, generalized, name-based information store. Mission designers first create a new store for a name (e.g. a player's name - "Vandal"). Once a store has been created for that name, data can be stored and retrieved inside that store by using 'paths' that lead to different information containers for that name. There is no limit to the number of containers that designers can add to a store, provided that all paths are named differently.

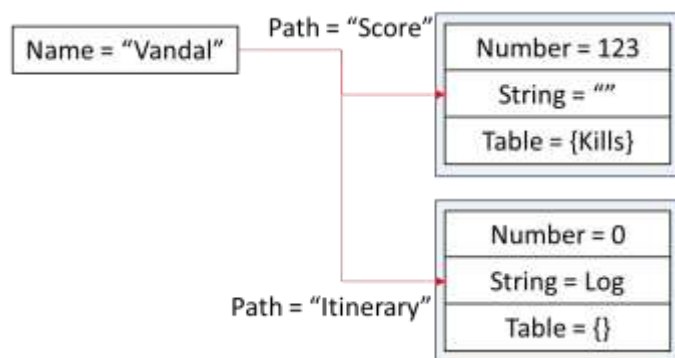
Inside each container, nameStats holds three separately accessible slots: one each for

- a number
- a string
- a table

Therefore, each path leads to a container that holds one instance of a string, a number and a table.

It's up to the designer which of these instances they use; they can use any one, two, or all three of them simultaneously.

The example on the right shows a store for the name "Vandal". This store currently is home to two containers: one that is accessed by the "Score" path, and one for the "Itinerary" path.



The container that is reached via the "Score" path uses both the number and table slots, but does not use the string slot. The number slot contains the current score for Vandal as a number (123), while the table slot contains a link to a table of kills (which is not further detailed in this demo)

The "Itinerary" path for Vandal leads to a container that uses the string slot to store a log of all take-offs and landings, while it currently doesn't use the number and table slots.

Note that above example illustrates how the information inside nameStats is organized. It's up to the designer to define how to use it. nameStats simply provides a simple, easy and unified accessor methods.

#### How to use nameStats

Since nameStats is designed for easy, safe use with numbers, strings and tables for a very specific set of goals (mission design), it's API is minimal, yet highly effective. Whenever you try to access something that does not yet exist, the structures required for access are created, and default values are returned.

This means that mission designers can always simply read or add information to named stores, and do not have to worry about allocating stores or containers.

nameStats centers around named stores because it assumes that mission designers want to store information based on players, and that each store is therefore similarly organized. All information inside a store should therefore be exclusive to that player.

### Numbers, Strings and Table Data inside a container

namedStore always provides three slots per container: a string, a number, and a table. For these slots, the API provides optimized accessors to accomplish the most common tasks:

- for numbers to set and change a number
- for strings to add to the current string (i.e. logging is supported out-of-the-box)
- for tables to store and retrieve (i.e. arbitrary user data under the mission designer's control)

So, for let us assume that a player in unit thePlane just landed at theAirfield. Let us further assume that we want to log this and the plane type under the "Itinerary" path in the players store. The entire code for this could look like this:

```
local pName = thePlane:getPlayerName()
local pType = thePlane:getTypeName()
local airfield = theAirfield:getName()
local aString = "\nLanded " .. pType .. " at " .. airfield
nameStats.addString(pName, aString, "Itinerary")
```

The addString method first looks if a store for pName exists. If not, a store is created. Then, nameStats looks for a container under path. Again, if none exists, a container is created. Finally, aString is added to the string that is currently in the string slot, creating a log.

### Example

So why this peculiar structure? Because a named store with pathed containers allows mission designers to easily implement otherwise complex tasks with ease. Let us imagine that a designer wants to track which player in a MP game kills how many planes of what type.

```
local theType = theKilledUnit:getTypeName() - type is path
nameStats.changeValue(thePlayerName, 1, theType) - add one
```

is all that is needed to do to track all kills (for each killed unit nameStats automatically opens a new container) – provided above lines are invoked for each kill.

At any time, we can iterate, and display all kills (per unit type and total) up to now for this player with

```
Local allKills = nameStats.getAllPaths(thePlayerName)
local totalK = 0
for idx, unitType in pairs(allKills) do
```

```

    local kills = nameStats.getValue(thePlayerName, unitType)
    totalK = totalK + kills
    trigger.action.outText(unitType .. ": " .. kills, 30)
end
trigger.action.outText( "Total Kills: " .. totalK, 30)

```

#### **Note:**

nameStats contains most of the required logic to build tree-based structures (node-leaf concept using the optional rootNode parameters). Since this isn't required for most missions, nameStats has been optimized for simple look-up-based access. Designers can easily extend nameStats to fully support tree structures.

#### *3.6.5.2 Dependencies*

None. This script is stand-alone and can be added to any mission without requiring any other scripts.

#### *3.6.5.3 Module Configuration*

None.

#### *3.6.5.4 API*

### **Numbers**

#### *3.6.5.4.1 `getValue(name, path)`*

Returns the number that is currently stored at path for store name. If this is a new name/path combination, 0 is returned.

#### *3.6.5.4.2 `changeValue(name, delta, path)`*

Adds delta to the value found at the name/path combination. Use negative numbers to subtract from the current value

#### *3.6.5.4.3 `setValue(name, newVal, path)`*

Sets the number at name/path to newVal.

### **Strings**

#### *3.6.5.4.4 `getString(name, path)`*

Returns the string that is currently stored at path for store name. If this is a new name/path combination, "" (an empty string) is returned.

3.6.5.4.5 `addString(name, aString, path)`

Adds aString to the string found at the name/path combination.

3.6.5.4.6 `setString(name, aString, path)`

Sets the contents of the string at name/path to aString

## Tables

3.6.5.4.7 `setTable(name, path, aTable)`

Sets the table at name/path to aTable

3.6.5.4.8 `getTable(name, path)`

Returns the table at name/path. Default table is {}

## General Name Store Management

3.6.5.4.9 `getAllNames()`

Returns a table of the names for all stores that currently exist in nameStats

3.6.5.4.10 `getAllPaths(name)`

Returns a table of all paths that are defined for name

3.6.5.4.11 `reset(name, path)`

Resets the three slots in a container for name/path to 0 (Zero, number slot), "" (empty string for string slot) and {} (empty table, table slot)

## 3.6.5.5 *ME Attributes*

nameStats requires no ME integration

## 3.6.5.6 *Using the module*

Copy the nameStats source into a DOSCRIPT action that runs at the start of the mission

### 3.6.6 cargoSuper (Lua Only)

cargoSuper is an API-only layer for managing cargo inventories/manifests. This module is primarily used by other modules to manage a unit's cargo weight and provide inventories for entire warehouses.

#### 3.6.6.1 Description

cargoSuper is a collection of methods that focus on abstracting cargo items and weight for transport and storage purposes in DCS. Central to cargoSuper are abstract “**mass objects**” that represent Things, and “**manifests**” that track these Things.

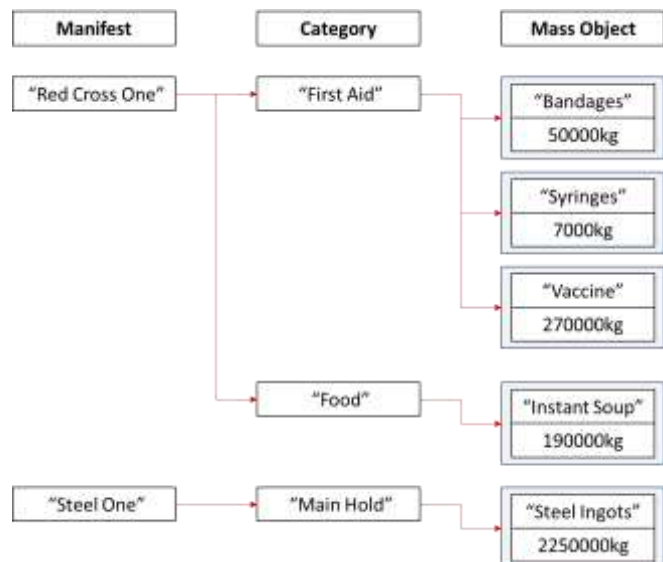
cargoSuper is primarily a book-keeping module: managing these mass objects is done simply by adding to, and removing them from manifests, and by tabulating them on demand. Manifests in cargoSuper are identified by their name; each manifest must have a unique name.

When a mass object is added to a manifest, it is always added under a ‘category’. Mission designers have free reign in naming these categories, and can elect to not use categories in manifests at all (in which case a default category name is substituted).

In the example to the right, we see two manifests that are maintained by cargoSuper: one each for the name “Red Cross One”, and one for “Steel One”.

In true bookie manner, cargoSuper does not care what these manifest names represent, except that they must be distinct from each other so cargoSuper can tell them apart.

If we look at the manifest for “Red Cross One”, we find that there are two entries for Category: “First Aid”, and “Food”. And in the “First Aid” category, we finally find the entries for the mass objects “Bandages” (50t), “Syringes” (7t) and “Vaccine” (270t).



The second category, “Food” contains a single mass object “Instant Soup” with 190t.

The second manifest, the one for “Steel One” contains only one category “Main Hold” that holds a single mass object “Steel Ingots” at 2250t. It is likely that such a configuration of manifests is used to track ship inventories; cargoSuper does not care, though, and mission designers are free to use the manifest/category/massObject system any way they choose.

It is important though, to remember that cargoSuper provides simple methods to add and remove them, as well as calculate total mass for entire manifests.

#### IMPORTANT:

cargoSuper does **not** modify a unit's cargo weight – it merely provides the book-keeping infrastructure so a mission designer can use these values to set a unit's cargo weight.

Mission scripts must invoke `trigger.action.setUnitInternalCargo()` with the calculated total mass of a manifest on their own (see workflow description below)

## cargoSuper workflow

cargoSuper is designed to support mission designers to help administrate cargo management for units and warehouses. cargoSuper manages mass objects (and their weight) by collecting them under manifests.

Basic workflow for using cargoSuper is as follows:

- **Creating mass objects (cargo)**

Before anything can be loaded into units or stored in warehouses, *massObjects* must be created using `createMassObject()`. This simply creates a mass object for inventory keeping; it does not load it into a unit, nor does it enter cargoSuper's manifest book-keeping

Mass objects can be created at any time. Note that although an important use for mass objects is to determine cargo mass, they are also used to simply represent stock. cargoSuper can also be used to provide full-blown inventory management for an entire warehouse, with the ability to determine mass only a nice addition.

Note that much like groups and units, all **mass objects must have a unique name**. If you create a mass object with a name that is already in use, you'll receive a warning message, and you may receive some unpredictable results. cargoSuper's API provides auto-naming features to automatically assign unique names for new mass objects. Since mass objects are often bound to in-mission objects, an easy way to assign a unique name to a mass object is by using the name from the mission object it represents.

- **Adding mass objects to manifests (add cargo to unit X)**

cargoSuper keeps tracks of mass objects and collects them under named "manifests".

Since unit names in DCS must be unique, it is a convenient way to collect all cargo into a manifest under that unit's name. So to add a mass object to unit X's manifest, we first get that unit's name with `X.getName()`, and then invoke `addMassObjectTo()` with the unit's name, a category (e.g. "first aid") and the mass object. This adds the mass object to cargo super's book-keeping into the manifest under unit X's name and the "first aid" category.

So what happens if you tell cargoSuper to add a mass object to a manifest that does not exist? Don't worry: cargoSuper simply opens a new manifest for the name that you supply, and adds the mass object.

- **(Setting a unit's cargo)**

Since cargoSuper is a general purpose-cargo inventory module, it does not set a unit's cargo by itself. It is up to the mission to set the unit's internal cargo mass by first inquiring the total mass of the manifest via `calculateTotalMassFor()`, and then invoking `trigger.action.setUnitInternalCargo()` with the new total mass value



- **Removing mass object from a manifest (remove cargo from unit X)**  
You can remove a mass object from a manifest at any time simply by invoking `removeMassObjectFrom()` for a manifest.
- **(Setting the unit's new cargo)**  
If you are managing the cargo weight for a unit, after removing the mass object from the manifest, it's time to set the unit's new cargo weight the same as before: calculate total mass via `calculateTotalMassFor()` and then invoke with the new value `trigger.action.setUnitInternalCargo()`.
- **Checking the inventory/manifest**  
At any time, you can check what is stored under a named manifest. This is a two-step process: since all mass items are stored in a manifest in categories, you first get all existing categories in a manifest with `getAllCategoriesFor()`, and then iterate all categories to get all mass objects stored in each category with `getManifestForCategory()`. Of course, mission designers can short-circuit this by not using categories and omit the first step.
- **Resetting manifests**  
`cargoSuper` provides a simple method to remove all mass objects from a cargo with `removeAllMassForCargo()` and even an entire manifest with `removeAllMassFor()`

**Note:**

Manifests keep track of mass objects as they are passed to them, and these mass objects are not assumed to be exclusive to a single manifest. This means that if a mission requires mass objects to be only tracked with a single manifest, the mission designer must ensure that a mass object is removed from one manifest when it is added to another,

### 3.6.6.2 Dependencies

`cargoSuper` requires `dcsCommon` and `nameStats`

### 3.6.6.3 Module Configuration

None.

### 3.6.6.4 API

## Managing Mass Objects

### 3.6.6.4.1 `createMassObject(massInKg, massName, referenceObject)`

Returns a new mass object with weighing `massInKg` and named `massName`. If `massName` is nil, `cargoSuper` generates a unique name for the object. `referenceObject` can be anything that is relevant from the mission designer's perspective. It often is a cargo unit created in DCS.

#### 3.6.6.4.2 `deleteMassObject(massObject)`

INTERNAL USE. This method deletes `massObject` from `cargoSuper`'s internal book keeping table, so it will no longer be checked against for name uniqueness. It does NOT remove the mass object from any manifest.

#### 3.6.6.4.3 `addMassObjectTo(name, category, theMassObject)`

Adds the `theMassObject` under `category` to a manifest named `name`. If the manifest does not exist, a new one is created. If `category` is `nil`, `cargoSuper` uses an internal default name.

#### 3.6.6.4.4 `removeMassObjectFrom(name, category, theMassObject)`

removes the `theMassObject` from `category` in `name`. If no such mass object exists in `category`, the command is ignored. If `category` is `nil`, an internal value is used as default.

### Managing Manifests

#### 3.6.6.4.5 `getAllCategoriesFor(name)`

returns a table of all currently defined categories in manifest `name`. This is commonly used to prepare to iterate through all categories.

#### 3.6.6.4.6 `removeAllMassForCategory(name, category)`

removes all mass objects from manifest `name` in `category`. If `category` is `nil`, an internal default is used.

#### 3.6.6.4.7 `removeAllMassFor(name)`

removes all mass objects from the entire manifest `name`

#### 3.6.6.4.8 `getManifestForCategory(name, category)`

returns all mass objects in `category` from manifest `name`. If `category` is `nil` then an internal default is used instead.

### Calculating Mass

#### 3.6.6.4.9 `calculateTotalMassForCategory(name, category)`

returns (in kg) the sum of weight for all mass objects in `category` of manifest `name`. If `category` is `nil`, an internal default value is used.

#### 3.6.6.4.10 `calculateTotalMassFor(name)`

returns (in kg) the sum of all weight for all mass objects in manifest `name`

#### 3.6.6.5 *ME Attributes*

None.

#### 3.6.6.6 *Using the module*

Copy the cargoSuper source into a DOSCRIPT action that runs at the start of the mission

### 3.6.7 cargoManager (Lua Only)

CargoManager is an API-only layer that generates **cargo events** for other scripts and modules. In a nutshell, your scripts pass it objects (e.g. cargo objects) to watch, and cargoManager invokes callbacks when something interesting happens to the watched object.

#### 3.6.7.1 Description

Essentially, your scripts first subscribe to be notified of cargo events, and then pass objects to watch for typical cargo events. Other modules, like for example cfxObjectSpawnZones can pass cargo automatically to cargo manager on spawning the object.

cargoManager checks each object once per second for changes in status. When an event happens to a watched object, cargoManager then invokes all callbacks with the necessary information

cargoManager is mainly used in scripts to detect when a helicopter picks up or puts down cargo objects, since DCS currently does not support generic events for this.

#### Cargo Events

“Cargo Events” are synthetic events, that cargoManager derives from monitoring the watched object’s state. If the state changes in certain ways, it generates an event and invokes all registered callbacks with event information:

- *lifted*  
When the cargo was previously deemed ‘grounded’, if now moved more than a meter, and is now considered ‘lifted’, i.e. picked up by a unit and being transported.
- *grounded*  
cargo hasn’t moved for a while, and is now considered ‘grounded’
- *disappeared*  
cargo object has disappeared
- *dead*  
cargo object was destroyed
- *new*  
a new cargo object was added to the watch pool,
- *remove*  
a cargo object was removed from the watch pool. Is invoked directly after a dead or disappeared event

#### Callback

When your script subscribes to cargo events, it must provide a callback that matches following profile

```
cargoEventCB(event, object, name)
```

with the following parameters

- `event`  
a string that contains one of the previously defined cargo events
- `object`  
the cargo object in question. Warning: the object may no longer be valid (in the case of a disappeared event)
- `name`  
the name of the cargo object. **Note:** if the cargo object itself is no longer valid (e.g. after a disappear event), the name still **is** valid.

### 3.6.7.2 Dependencies

CargoManager requires `dcsCommon` and `cfxZones`

### 3.6.7.3 Module Configuration

`cfxCargoManager.upd` – updates per second. Set to 1

### 3.6.7.4 API

#### 3.6.7.4.1 `addCallback(cb)`

Adds the callback `cb` to list of callbacks to invoke when a cargo event happens.

#### 3.6.7.4.2 `getCargoStatusFor(theCargoObject)`

Returns the current state (a string, e.g. “lifted”) for the `theCargoObject`. If the cargo object does not exist, `nil` is returned..

#### 3.6.7.4.3 `addCargo(theCargoObject)`

Adds the `theCargoObject` (a DCS object) to the pool of watched objects. You must ensure that this is a proper static object, or the results can be unpredictable.

#### 3.6.7.4.4 `removeCargoByName(cargoName)`

removes a cargo object with the name `cargoName` from the pool of watched cargo objects.

#### 3.6.7.4.5 `removeCargo(theCargoObject)`

removes the `theCargoObject` (a DCS object) from the pool of watched cargo objects

#### 3.6.7.4.6 `getAllCargo(filterForState)`

Returns a table of all cargo objects currently in the watch pool. If `filterForState` is given (not `nil`), the list is filtered for all objects that match the string given in `filterForState` (e.g. “lifted”)

#### 3.6.7.5 *ME Attributes*

None.

#### 3.6.7.6 *Using the module*

Copy the cfxCargoManager source into a DOSCRIPT action that runs at the start of the mission

### 3.6.8 cfxGroundTroops (Lua Only)

#### 3.6.8.1 Description

Technically, this module sits between Foundation, and Enhancements **It manages** groups that have been issued **orders**. All interaction with cfxGroundTroops is via API; many modules higher up in the hierarchy make heavy use of this module (spawners, owned zones, etc.). For example, cfxSpawnZones create groups, issue them orders, and then pass them off to cfxGroundTroops for management.

#### Orders

As described under Important Concepts: → Orders, orders are central to DML, and cfxGroundTroops is the module that manages orders for ground troops. Currently, the following orders are recognized (meaning that any module that requests an order attribute will automatically support them).

- **guard**  
Orders to guard means that the group is to remain in place, and attack any enemy forces in the vicinity. The range parameter determines the engagement radius
- **attackOwnedZone**  
Look for, and then move towards the closest non-owned zone managed by the cfxOwnedZone module. Engage any enemy troops there. Once the zone has been conquered, move to the next. When all zones have been conquered, switch to “guard”.
- **attackZone**  
Move toward a named zone. Attack any enemy troops on the way.
- **lase/laze**  
Stay in place and laser-designate any enemy troops that come in range and have a direct LOS. The range parameter determines how far the group can “see”. Each group only lases one target.
- **train/training/dummy/dummies**  
Do nothing, do not engage, weapons tight.

#### Performance considerations

Although lightweight, cfxGroundTroops *can* have significant impact on a mission’s performance – although this is usually caused by the sheer number of troops in-game rather than the processing time required to manage the troop’s orders. If you experience performance issues, you may want to first see how many groups your mission is spawning, and then how cfxGroundTroops is managing them, with particular attention to the update intervals you have configured.

#### Troop Management Models

GroundTroops supports multiple troop order management models, controlled by the configuration settings:

- *Standard*  
In this mode, all groups are inspected every update pass. This burns performance and can result in performance spikes. It can also overload DCS's order AI when it receives too many new orders simultaneously, so use this update model only when you are using few managed groups
- *Queued*  
In this mode, every update pass, one exactly one group is processed. Since ground units progress on a sedate pace, it's enough to check and update every unit once every few minutes. This method produces an even load with excellent response to most tactical situations
- *Scheduled*  
In this mode, updates per group are scheduled individually per group. This mode distributes workload evenly and is future-proof for when DCS introduces multi-processing for Lua scripts.

## Callbacks

cfxGroundTroops supports a callback infrastructure that scripts can subscribe to and be notified of certain events. To have your method invoked, use `addTroopsCallback()` with your callback handler as parameter

The callback method must adhere to the following profile

```
theCallback(reason, theGroup, theOrders, data)
```

with `reason` being a string describing why the callback occurred, `theGroup` being the affected group, `theOrders` a string containing the groups current orders, and `data` being reason-specific data, with `data.troops` always containing a reference to the Troop table for this group

The callbacks can be invoked with the following reasons:

- “arrived”  
theGroup has arrived at the destination. Group orders will be switched to “guard” automatically.
- “dead”  
The entire group was destroyed. Group is automatically removed from managed pool.
- “neutralized”  
An enemy group was successfully destroyed by a group with “guard” orders. The attribute `data.enemy` contains the destroyed group.
- “engaging”  
An enemy group is being engaged by a group with “guard” orders. The attribute `data.enemy` contains the group that is being engaged.
- “lase:tracking”  
A group with orders to lase has found a target and started tracking an enemy unit. `data.enemy` contains the enemy unit, and `data.tracker` the unit that is doing the tracking.



- “lase:stop”  
A group with orders to lase has stopped tracking the target units. Reasons for that can be multiple: the tracked unit has moved out of LOS, is dead, or the tracking unit has died.

## Managed Capacity

In addition to the order update loop, GroundTroop allows you to limit the number of groups it actively manages. By setting `cfxGroundTroops.maxManagedTroops` you can turn on managed capacity. When setting this value to any positive number, the pool of managed troops is limited that number. Any new troops that are added to the pool will instead be added to queues (separate for faction to ensure that no faction gains an advantage by flooding the pool). Only when troops are removed from the pool, it is replenished from the queues. Troops that are queued simply aren't order-managed. They will still fight should they encounter enemies and will follow any commands they have been issued prior to being added to the pool.

Place a cap on the pool when you experience crashes while your mission has deployed many groups. From experience, DCS can crash if too many groups are being issues orders in a short time span, which can happen when OwnedZones change hands (many groups receive new routing orders) – especially if you choose to have groups follow roads (which is an option when giving orders)

## Troops vs DCS Groups

`cfxGroundTroops` uses the concept of a ‘Troop’ wrapper (see Troop Table, below) that encapsulates a DCS group (i.e. a group as returned from DCS) of ground units that it manages with additional data. In simple terms, Troops are “a DCS group plus Orders”. Please refer to the section on Orders to read up on the details. To have GroundTroops manage a group of ground units, they first have to be converted into a “troop” structure by invoking `cfxGroundTroops.createGroundTroops()`. The returned table can then be modified, and finally submitted to the pool of managed troops by invoking `cfxGroundTroops.addGroundTroopsToPool()`. From this point on, `cfxGroundTroops` monitors and manages the group. To remove a group from the pool, you can invoke `cfxGroundTroops.getGroundTroopsForGroup()` to retrieve the troop wrapper, and then invoke `cfxGroundTroops.removeTroopsFromPool()` to have them removed. This will not change the current groups orders nor remove them from the mission – they merely are no longer managed by `cfxGroundTroops`.

## The Troop Table

`cfxGroundTroop` wraps a DCS group with a table to contain additional data for order management. Scripts can access this information, but caution is advised when changing data in the troop table: this can have unpredictable results. While most of the fields are internal, the following attributes may be of interest:

- `group`  
the DCS group that this troop table wraps.

- `orders`  
The orders (a string, e.g. "lase") for this group. This is `cfxGroundTroops` main entry point for handling the group during update. Modifying this attribute will take effect in the next update cycle.
- `coalition`  
The coalition the group belongs to. Simply a copy from `group:getCoalition()`
- `name`  
The name of group. Cached in case group becomes inaccessible. Persists past `group:delete()` so you can still access the group's name
- `range`  
value of the range parameter as used by some orders. Changes here take effect in the next update cycle
- `destination`  
Only defined for some orders. A `cfxZone` describing the destination area
- `enemy`  
Only defined for some orders. A DCS Group that this group is tasked with engaging.

### 3.6.8.2 Dependencies

**Required:** `dcsCommon`, `cfxZones`, `dcsCommander`

**Optional:** `cfxOwnedZones`

### 3.6.8.3 Module Configuration

`cfxGroundTroops` can use a configuration zone for setting up main options. To configure `cfxGroundTroops` via a configuration zone,

- Place a Trigger Zone in ME anywhere
- Name it "groundTroopsConfig" (note: name must match exactly)
- Add any of the following attributes to this zone:

Name	Description
<code>queuedUpdates</code>	set to true to process one group per turn. To work this way, <code>scheduledUpdates</code> (see below) must be false. Default is false
<code>scheduledUpdates</code>	set to false to allow queued of standard updates. Overrides <code>queuedUpdates</code> is true. Defaults to false
<code>maxManagedTroops</code>	Defines a cap on the number of actively managed groups ( <i>not</i> units!). Once the cap is exhausted, new groups are placed into a smart managed queue and fed into the managed pool when slots become available. Queued troops will still engage enemies when sighted, but not move nor lase enemies. Set to <b>-1</b> for an infinite cap. Defaults to 65

Name	Description
monitorNumbers	Set to true do receive debugging info when queuing and dequeuing groups. Defaults to false
standardScheduleInterval	Interval (in seconds) between updating managed queue. Defaults to 30 seconds (twice a minute)
guardUpdateInterval	Interval (in seconds) between updates for groups with “guard” orders. Defaults to 30 seconds.
trackingUpdateInterval	Update interval for groups with “lase” orders that track moving vehicles. Defaults to 0.5 (twice per second)

### 3.6.8.4 API

#### 3.6.8.4.1 `addTroopsCallback(theCallback)`

Adds theCallback to the list of callbacks to invoke when cfxGroundTroops detects a troop event. The callback must match the profile

```
theCallBack(reason, theGroup, theOrders, data)
```

with reason being a string that describes the reason for the callback (see above), theGroup being the group that triggered the event, theOrders the group’s current orders, and data the event-specific additional data (if defined).

#### 3.6.8.4.2 `createGroundTroops(inGroup, range, orders)`

returns the Troop Table for inGroup that can be managed with cfxGroundTroops. The parameter range is order-dependent, and orders is the order string (described in →Orders )

#### 3.6.8.4.3 `addGroundTroopsToPool(troops)`

Adds the TroopTable troops to cfxGroundTroops’ pool of managed troops.

#### 3.6.8.4.4 `removeTroopsFromPool(troops)`

Removes the TroopTable troops from cfxGroundTroops’ pool of managed troops. Note that this is done automatically when a managed group is destroyed; you only need to use this if you want to prevent cfxGroundTroops from managing the group described in troops.

#### 3.6.8.4.5 `isDeployedGroundTroop(aGroup)`

Returns true if aGroup is managed by cfxGroundTroops (or waiting in a queue to be managed)

#### 3.6.8.4.6 `getGroundTroopsForGroup(aGroup)`

Returns the Troop Table for aGroup, provided aGroup is being managed by cfxGroundTroops.

### 3.6.8.5 ME Attributes

No ME interface

#### 3.6.8.6 *Using the module*

Copy the cfxGroundTroops source into a DOSCRIPT action that runs at the start of the mission

Add a configuration zone with ME to override default configuration.

### **3.6.9 cfxGroups (Lua Only)**

#### *3.6.9.1 Description*

This low-lever module primarily supplies information about player slots. This information is usually only required for missions that in one way or another need to block slots.

#### *3.6.9.2 Dependencies*

#### *3.6.9.3 Module Configuration*

#### *3.6.9.4 ME Attributes*

#### *3.6.9.5 API*

#### *3.6.9.6 Using the module*

cf/x Dynamic Mission Library  
for DCS

## PART III: FOUNDATION API

## 4 Foundation API

This section contains a detailed, comprehensive reference of those modules that have extensive API:

- dcsCommon – the bedrock foundation. Mission designers use it in all their scripts.
- cfxPlayer – the module that creates player events and maintains a DB of player units and players
- cfxZones – the module for ME integration and intelligent zone management

Note that this Part contains information that is **only necessary for mission designers that want to access DML via Lua**. If you do not intend to write Lua scripts yourself, you can safely skip this section.

## 4.1 dcsCommon API

This is the API for the foundation's most fundamental module, dcsCommon.

### 4.1.1 Miscellaneous Methods

Miscellaneous methods that are helpful for odd things like random, headings, conversions etc.

#### 4.1.1.1 *libCheck(testingFor, requiredLibs)*

returns true if all modules listed in requiredLibs are defined

#### 4.1.1.2 *smallRandom(theNum)*

Returns a random number. Useful for random numbers lower than 50 since DCS's random generator is based on Perlin, which returns a sequence of random numbers that are random on average, but close together.

#### 4.1.1.3 *randomDegrees()*

returns a random number between 0 and 359

#### 4.1.1.4 *randomPercent()*

returns a random number between 0 and 100

#### 4.1.1.5 *randomPointOnPerimeter(sourceRadius, x, z)*

returns a random point (xr, 0, zr) on the circle placed at (x, 0, z) with radius sourceRadius

#### 4.1.1.6 *randomPointInCircle(sourceRadius, innerRadius, x, z)*

returns a random point (xr, 0, zr) inside a circle located at (x, 0, z) with sourceRadius, and that is at least innerRadius distant from center (x, 0, y)

#### 4.1.1.7 *distFlat(p1, p2)*

Returns the 'flat' distance (distance as measured on a map, ignoring any height) between two points p1 and p2. Input points 3D (x, y, z)

#### 4.1.1.8 *dist(point1, point2)*

Returns distance between two 3D points poin1 and point2

#### 4.1.1.9 *delta(name1, name2)*

Returns the distance between two objects named name1 and name2



#### 4.1.1.10 *dcsCommon.lerp(a, b, x)*

Returns a value between a and b, with x indicating 'how far from a towards b' in percent.  
Example: `lerp(10, 20, 0.5)` is the value 50% percent between 10 and 20, returning 15

#### 4.1.1.11 *dcsCommon.bearingFromAtoB(A, B)*

Returns the bearing (in rad) A lies in relation to B (looking from B)

#### 4.1.1.12 *bearingInDegreesFromAtoB(A, B)*

Returns the bearing (in degrees) A lies in relation to B (looking from B)

#### 4.1.1.13 *compassPositionOfARelativeToB(A, B)*

Returns the compass position ("N", "NW", etc) A lies in relation to B (looking from B)

#### 4.1.1.14 *clockPositionOfARelativeToB(A, B, headingOfBInDegrees)*

Returns the Clock Position (12, 1, 2, ...) of A in relation to B (looking from B)

#### 4.1.1.15 *getClockDirection(direction)*

Returns o'clock for a direction (direction in degrees)

#### 4.1.1.16 *topClone(orig)*

Returns a shallow clone of table orig

#### 4.1.1.17 *clone(orig)*

Returns a deep clone of table orig

#### 4.1.1.18 *rotatePointAroundOrigin(inX, inY, angle)*

Returns px and py after being rotated angle around origin

#### 4.1.1.19 *bool2Text(theBool)*

Returns "true" or "false", depending on the value of theBool

#### 4.1.1.20 *bool2YesNo(theBool)*

Returns "yes" or "no" depending on the value of theBools

#### 4.1.1.21 *dumpVar(key, value, prefix, inrecursion)*

Dumps the entire contents of value to the log. Recursively dumps all table elements, including subtables. To dump the contents of the variable x to the log, invoke

```
dcCommon.dumpVar("this is x:", x)
```

Inspect the DCS log for results, search for "this is x:" to find the location

#### *4.1.1.22 dumpVar2Str(key, value, prefix, inrecursion)*

returns a string that contains the entire contents of value as text. Recursively walks through all table elements, including subtables. To get the contents of the variable x into s, invoke

```
local s = dcCommon.dumpVar2Str("x is:", x)
```

#### *4.1.1.23 event2text(id)*

Translates the DCS event id into human readable string

#### *4.1.1.24 smokeColor2Text(smokeColor)*

Translates a DCS smoke color into human readable string (e.g. "RED")

#### *4.1.1.25 smokeColor2Num(smokeColor)*

Translates human-readable smoke color (e.g. "red") into a color that DCS understands

#### *4.1.1.26 markPointWithSmoke(p, smokeColor)*

Places a smoke colored smokeColor at the location p. The smoke origin is height-adjusted for the terrain.

### 4.1.2 Table / String Managements

These methods are specialized to assist in manipulating arrays and strings

#### 4.1.2.1 *getSizeOfTable(theTable)*

Returns the size (number of elements) of any table (including, but not limited to, arrays)

#### 4.1.2.2 *dcsCommon.findAndRemoveFromTable(theTable, theElement)* (assumes array)

Looks for, and then removes theElement from the array theTable

#### 4.1.2.3 *pickRandom(theTable)* (assumes array)

Returns a random element from the array theTable

#### 4.1.2.4 *enumerateTable(theTable)*

converts the arbitrary array/table theTable to an array, dropping any keys.

#### 4.1.2.5 *arrayContainsString(theArray, theString)*

returns true if the array theArray contains an element that is equal to theString

#### 4.1.2.6 *splitString(inputstr, sep)*

Returns an array that contains all parts of inputstr, split at the separator sep. For example

```
dcsCommon.split("hello my friends", " ")
```

returns {"hello", "my", "friends"}. If no sep is given, blank (" ") is used

#### 4.1.2.7 *trim(inputstr)*

Returns the contents of inputstr with leading and trailing blanks (" ") removed.

#### 4.1.2.8 *trimArray(theArray)*

Returns an array based on theArray, in which all elements have their leading and trailing blanks (" ") removed.

#### 4.1.2.9 *stringStartsWith(theString, thePrefix)*

Returns true if theString starts with thePrefix, false otherwise

#### 4.1.2.10 *removePrefix(theString, thePrefix)*

Returns the contents of theString with thePrefix removed. If theString does not start with thePrefix. theString is returned unchanged.

#### *4.1.2.11 stringEndsWith(theString, theEnding)*

Returns true if theString ends with theEnding, false otherwise

#### *4.1.2.12 removeEnding(theString, theEnding)*

Returns the contents of theString with theEnding removed. If theString does not end with theEnding. theString is returned unchanged.

#### *4.1.2.13 containsString(inString, what, caseSensitive)*

Returns true if the string inString contains the value of what. If caseSensitive is true, the comparison is exact, else the case is ignored.

#### *4.1.2.14 numberUUID()*

Returns a unique number for each time it is invoked

#### *4.1.2.15 uuid(prefix)*

Returns a string that starts with the value of prefix, with a unique number appended each time it is invoked.

### 4.1.3 Vector Math

These methods implement common vector methods for DCS points (vec3 that are based on "x", "y", "z")

#### 4.1.3.1 *vAdd(a, b)*

Returns the sum of a and b

#### 4.1.3.2 *vSub(a, b)*

Returns  $a - b$

#### 4.1.3.3 *vMultScalar(a, f)*

Returns a multiplied by the scalar f ("vector a times number f")

#### 4.1.3.4 *vLerp(a, b, t)*

Returns the point x that is t percent between a and b

#### 4.1.3.5 *mag(x, y, z)*

Returns the magnitude of an implicit vector made from (x, y, z)

#### 4.1.3.6 *vMag(a)*

Returns the magnitude of vector/point a

#### 4.1.3.7 *magSquare(x, y, z)*

Returns the squared magnitude of a vector comprising of (x, y, z). magSquare is slightly faster than mag because it does not calculate the square root.

#### 4.1.3.8 *vNorm(a)*

Returns a in normalized form, i.e. the magnitude of vNorm(a) is 1.0 while it still points in the exact same direction as a.

#### 4.1.3.9 *dot(a, b)*

Returns the dot product of the vectors a and b (essentially b projected onto a)

#### 4.1.4 Airfield, Landable Ships and FARP

These methods simplify access to airfields, ships that aircraft can land on, and FARPs

##### 4.1.4.1 *getAirbaseCat(aBase)*

Returns the category of airbase aBase

##### 4.1.4.2 *getFirstFreeParkingSlot(aerodrome, parkingType)*

Returns the first unoccupied parking slot with correct type of aerodrome. If no parkingType is specified, the first free slot is returned.

##### 4.1.4.3 *getAirbasesInRangeOfPoint(center, range, filterCat, filterCoalition)*

Returns an array of all airbases that are inside a circle with a radius of range around the coordinates center. If filterCat is given, only airbases of that category are returned. If filterCoalition is specified, only airbases that are owned by that coalition are returned.

##### 4.1.4.4 *getAirbasesInRangeOfAirbase(airbase, includeCenter, range, filterCat, filterCoalition)*

Returns an array of all airbases that are inside a circle centered on airbase with radius range. If includeCenter is true, the result includes airbase. If filterCat is given, only airbases of that category are returned. If filterCoalition is specified, only airbases that are owned by that coalition are returned.

##### 4.1.4.5 *getAirbasesInRangeOfAirbaseList(theCenterList, includeList, range, filterCat, filterCoalition)*

Returns an array of all airbases that in range of all the airbases specified in the table theCenterList. If includeList is true, the result includes the airbases from theCenterList. If filterCat is given, only airbases of that category are returned. If filterCoalition is specified, only airbases that are owned by that coalition are returned.

##### 4.1.4.6 *getAirbasesWhoseNameContains(aName, filterCat, filterCoalition)*

Returns a list of all airbases on the map whose name contains aName. If filterCat is given, only airbases of that category are returned. If filterCoalition is specified, only airbases that are owned by that coalition are returned.

##### 4.1.4.7 *getFirstAirbaseWhoseNameContains(aName, filterCat, filterCoalition)*

Returns the first airbase on the map whose name contains aName. If filterCat is given, only airbases of that category are returned. If filterCoalition is specified, only airbases that are owned by that coalition are returned.

#### 4.1.4.8 *getClosestAirbaseTo(thePoint, filterCat, filterCoalition)*

Retrurns the closest airbase to thePoint. If filterCat is given, only an airbase of that category is returned. If filterCoalition is specified, only an airbase that is owned by that coalition is returned.

### 4.1.5 Group handling

These methods simplify accessing and getting information on groups. Usually, methods require a DCS group as input

#### 4.1.5.1 *livingUnitsInGroup(group)*

Returns an array of all units of group that are alive

#### 4.1.5.2 *getClosestLivingUnitToPoint(group, p)*

Returns the living unit of group that is closest to point p

#### 4.1.5.3 *getClosestLivingGroupToPoint(p, coal, cat)*

Returns the group with living units that is closest to p and belongs to coalition coal and of category cat. If cat isn't specified, GROUND units are returned.

#### 4.1.5.4 *getLivingGroupsAndDistInRangeToPoint(p, range, coal, cat)*

Returns an array of {"group":group, "dist":dist} elements of all groups that are in range of point p and of category cat and belong to coalition coal. If no cat is specified, GROUND is used.

#### 4.1.5.5 *getGroupLocation(group)*

Returns the location of the first living unit of group. Group can be string or DCS group

#### 4.1.5.6 *getGroupUnit(group)*

Returns the first unit in group that is alive, or nil. Group can be string (name of group) or DCS group.

#### 4.1.5.7 *getFirstLivingUnit(group)*

Alias for getGroupUnit() with a much better name

#### 4.1.5.8 *isGroupAlive(group)*

Returns true if group has at least one living unit. Group can be string or DCS group

#### 4.1.5.9 *getLiveGroupUnits(group)*

Returns an array of all living units of group

#### 4.1.5.10 *getGroupTypeString(group)*

Returns a string with the TypeNames of all living units inside the group. The TypeNames are separated by comma.



#### 4.1.5.11 *getGroupTypes(group)*

Returns an array of the TypeStrings of all living units in group

#### 4.1.5.12 *getGroupAvgSpeed(theGroup)*

Returns the average speed of all living units in theGroup

#### 4.1.5.13 *getGroupMaxSpeed(theGroup)*

Returns the speed of the currently fastest unit in theGroup

### 4.1.6 Unit Handling

These methods simply accessing a unit's properties

#### 4.1.6.1 *isSceneryObject(theUnit)*

Returns true if theUnit is a scenery object

#### 4.1.6.2 *isTroopCarrier(theUnit)*

Returns true if theUnit can carry infantry (currently a set number of helicopters. APC etc. currently aren't included)

#### 4.1.6.3 *getUnitAlt(theUnit)*

Returns theUnit's y component in meters

#### 4.1.6.4 *getUnitAGL(theUnit)*

Returns a units height above ground in meters

#### 4.1.6.5 *getUnitSpeed(theUnit)*

Returns theUnit's speed (in m/s)

#### 4.1.6.6 *getClosingVelocity(u1, u2)*

Returns the closing velocity (how fast the units approach each other) in m/s. A negative value means that they are separating at that speed.

#### 4.1.6.7 *getUnitHeading(theUnit)*

Returns the heading (in rad) of theUnit

*4.1.6.8 getUnitHeadingDegrees(theUnit)*  
Returns the heading (in degrees) of theUnit

*4.1.6.9 unitIsInfantry(theUnit)*  
Returns true if theUnit is an infantry unit

### 4.1.7 Spawning Units / Group, Routes, Tasks

These units simplify constructing and manipulating the tables (data blocks) that are used for spawning units/groups in DCS

#### 4.1.7.1 *createEmptyGroundGroupData (name)*

Returns an empty group data block for a group named name, that can be used later (after adding more data) to spawn groups. Defaults to ground troops.

#### 4.1.7.2 *createEmptyAircraftGroupData (name)*

Returns an empty group data block that can be used to create air groups. The group's name is name.

#### 4.1.7.3 *createAircraftRoutePointData(x, z, altitudeInFeet, knots, altType, action)*

Returns a route waypoint that can be used to assemble an air group's path. The waypoint is located at (x, altitudeInFeet,z) and is of type action, with a speed of knots.

#### 4.1.7.4 *addRoutePointDataToRouteData(inRoute, x, z, altitudeInFeet, knots, altType, action)*

Adds a new air waypoint to inRoute at location (x, altitudeInFeet, z) with a speed of knots and action.

#### 4.1.7.5 *addRoutePointDataToGroupData(group, x, z, altitudeInFeet, knots, altType, action)*

Adds a new air waypoint to an air group data block at location (x, altitudeInFeet, z) with a speed of knots and action. (Simplified accessor so you don't have to construct aircraft routes yourself). Will add a route to the data block if none present

#### 4.1.7.6 *addRoutePointForGroupData(theGroup, theRP)*

Adds route point theRP to data block theGroup. (air???)

#### 4.1.7.7 *createEmptyAircraftRouteData()*

Returns an empty data block that is used to assemble aircraft routes

#### 4.1.7.8 *createTakeOffFromParkingRoutePointData(aerodrome)*

Returns a route point for an aircraft to take off from aerodrome

#### 4.1.7.9 *createOverheadAirdromeRoutePointData(aerodrome)*

Returns a route point overhead aerodrome

#### 4.1.7.10 *createLandAtAerodromeRoutePointData(aerodrome)*

Returns a route point to land at aerodrome

#### 4.1.7.11 *createRPFormationData(findex)*

returns a route point that specifies formation findex

#### 4.1.7.12 *addTaskDataToRP(theTask, theGroup, rpIndex)*

Adds theTask to route point rpIndex of theGroup

#### 4.1.7.13 *createPayload(fuel, flare, chaff, gun)*

Returns a payload data block to be used for spawning aircraft

#### 4.1.7.14 *createCallsign(cs)*

Returns a callsign data block to be used for spawning

#### 4.1.7.15 *createGroundUnitData(name, unitType, transportable)*

Returns a ground unit data block for a unit named name and of TypeString unitType. This data block is then added to groups. If transportable is true, the unit can be transported by other units capable of transporting units.

#### 4.1.7.16 *createAircraftUnitData(name, unitType, transportable, altitude, speed, heading)*

Creates a data block for an air unit name of unitType – WTF are the other params? Check code!!! TODO: check this

#### 4.1.7.17 *addUnitToGroupData(theUnit, theGroup, dx, dy, heading)*

Adds theUnit to theGroup at an offset of dx, dy to the group's location, and facing heading.

#### 4.1.7.18 *createSingleUnitGroup(name, theUnitType, x, z, heading)*

Simplified method to create a single-unit group name with a single unit of theUnitType at location x, 0, z) and facing heading. Returns the unit's group

#### 4.1.7.19 *arrangeGroupDataIntoFormation(theNewGroup, radius, minDist, formation, innerRadius)*

Recalculate all unit's relative locations relative to the group's location, using radius, minDist, formation and innerRadius as parameters for formation

TODO: explain formations and params

*4.1.7.20 createGroundGroupWithUnits(name, theUnitTypes, radius, minDist, formation, innerRadius)*

returns a group data block named name from array theUnitTypes using formation and parameters. Group is always ground units.

TODO: explain formation and parameters

*4.1.7.21 createGroupDataFromLiveGroup(name, newName)*

Using all living units from the living (in-game, already spawned) group name, create a data block based on their data, and return that data block:

TODO: revisit code to look at what it really does.

*4.1.7.22 rotateGroupData(theGroup, degrees, cx, cz) – data block*

Rotates the group data block theGroup by degrees around a center point defined by (cx, 0, cz).

*4.1.7.23 offsetGroupData(theGroup, dx, dy)*

Offsets all unit's locations in data block theGroup by dx and dy

*4.1.7.24 moveGroupDataTo(theGroup, xAbs, yAbs)*

Update data in theGroup's data block to that the group's location is located at (xAbs, 0, yAbs). This will move all unit data as well

#### 4.1.8 Static Objects

These methods simplify creating static objects and placing them in-game

##### 4.1.8.1 *createStaticObjectData(name, objType, heading, dead, cargo, mass)*

Returns a generic data block to be used for spawning static objects. The properties set are name (must be unique), objType (the TypeString), the heading it is facing, dead (set to true if the destroyed variant of the model is to be used), cargo (set to true if it can be used as cargo) and mass (amount in kg)

##### 4.1.8.2 *createStaticObjectDataAt(loc, name, objType, heading, dead)*

Returns a data block for spawning static objects like above at the location loc, with loc being in the format (x, ignored, z). No options for setting dead, cargo or mass

##### 4.1.8.3 *linkStaticDataToUnit(theStatic, theUnit, dx, dy, heading)*

Link the data block for the static object to be created to theUnit. theUnit must be a ship. The parameters dx and dy describe the initial offset to theUnit's center. You must link a static object before you spawn it.

##### 4.1.8.4 *offsetStaticData(theStatic, dx, dy)*

Offset's the location inside the data block theStatic by the amount of dx and dy

##### 4.1.8.5 *moveStaticDataTo(theStatic, x, y)*

Sets the location information inside the data block theStatic to the position (x, 0, z)

##### 4.1.8.6 *createStaticObjectForCoalitionAtLocation(theCoalition, loc, name, objType, heading, dead)*

Create in-game and return a static object as described by the parameters.

THIS METHOD SPAWNS STATIC OBJECTS

##### 4.1.8.7 *createStaticObjectForCoalitionInRandomRing(theCoalition, objType, x, z, innerRadius, outerRadius, heading, alive)*

TODO: Verify

Spawns (creates in-game) and returns a static object belonging to theCoalition and that is described by the parameters inside a circle around the coordinates (x, 0, z) with the radius outerRadius and at least innerRadius distant to the center.

#### 4.1.9 Coalition

##### 4.1.9.1 *getEnemyCoalitionFor(aCoalition)*

Returns the coalition number (1 or 2) that is the enemy of aCoalition. Input can be 1, 2, "red" or "blue". If aCoalition is neither of those values, nil is returned (this means that neutral has no enemies!)

#### 4.1.9.2 *getACountryForCoalition(aCoalition)*

Returns the first country ID (a number) that is part of aCoalition (as set in the coalition builder in ME)

#### 4.1.9.3 *coalition2county(inCoalition)*

Returns the joint task force 'country' for inCoalition (0, 1, or 2)

### 4.1.10 Event Handling

#### 4.1.10.1 *addEventHandler(f, pre, post, rejected)*

Adds your event handler *f* to the list of methods that are to be invoked when an event happens. In addition to the handler *f*, you can also install three additional methods: *pre* is a method that is invoked with the unprocessed event to pre-process and determine if *f* should be invoked. If *pre* returns true, *f* will be invoked, *rejected* otherwise. *post* is a method that is invoked after *f* was invoked, and can be used to clean up any data prepared in *pre*. *post* is only invoked if *f* was invoked. If you don't specify any of *pre*, *post* or *rejected*, dcsCommon's internal (default) methods are invoked instead (they usually do nothing – unless you override them for your own purposes – be very careful when doing that, it is not recommended!). *addEventHandlers* returns the id for the event handler to be used with *removeEventHandler*

#### 4.1.10.2 *preCall(e)*

Default pre-processor for events, always returns true.

#### 4.1.10.3 *postCall(e)*

Default post-processor for events, does nothing.

#### 4.1.10.4 *addEventHandlerForEventTypes(f, evTypes, post, rejected)*

Adds your event handler *f* to the mission. *f* will be invoked for the events listed in the *evTypes* array. *Post* and *rejected* are methods that are invoked after *f* (post) or the event was filtered (rejected). Returns the id for the event handler to be used with *removeEventHandler*

#### 4.1.10.5 *removeEventHandler(id)*

Removes the event handler

## 4.2 cfxPlayer API

As a classic ‘Event Generator’, most interaction with cfxPlayer is via the callbacks, with cfxPlayer also providing a number of tables and convenience methods for accessing player units, player groups, or player info.

Missions in DCS mostly disregard players and focus on units and strategic considerations, making missions that focus on a player a rather difficult proposition. cfxPlayer helps in this regard, as it can synthesize player events by monitoring all players and the units they occupy. If your mission focuses on player actions, and your mission design includes players as key concepts, you will probably want to use cfxPlayer in your missions.

### 4.2.1 Tables

#### 4.2.1.1 *cfxPlayer.playerDB*

This contains a table indexed by unit name that is updated regularly with the units of all player-controlled units and the groups they belong to. Each table entry has the following attributes:

- **name**: Name of the **player’s unit**. THIS IS NOT THE PLAYER’S NAME
- **unit**: the unit that the player controls
- **unitName**: the name of the unit that the player controls. Same as **name**
- **group**: the group that the player’s unit belongs to
- **groupName**: the name of the group that the unit belongs to
- **coalition**: the coalition that this unit is aligned with

#### 4.2.1.2 *cfxPlayerGroups*

A table indexed by group name that contains information about all groups in-game that currently contain at least one player-controlled unit. Each table entry has the following attributes:

- **group**: theGroup
- **name**: the group’s name
- **primeUnit**: the first living unit that has is player-controlled
- **primeUnitName**: the name of the primeUnit
- **id**: the group ID

#### 4.2.1.3 *cfxPlayer.netPlayers*

A (key, value) dictionary of (player name, unit name). Note that the unit may no longer exist, and that a player may have left the mission.

### 4.2.2 Callback Handling

#### 4.2.2.1 *addMonitor(myCallback, eventsFilter)*

This adds the method *myCallback* to cfxPlayer’s list of function to invoke when it detects a player event. The **optional** table *eventsFilter* contains an array of events that lists the events *myCallback* should be invoked for. This is a subset of all events the cfxPlayer defines.



For example:

```
local eventFilter = {"newGroup", "removeGroup"}  
cfxPlayer.addMonitor(myCallback, eventFilter)
```

The following events are currently defined:

- "new" – new player-controlled unit appeared
- ("side") – currently never invoked
- ("group") – currently never invoked
- ("unit") – currently never invoked
- "leave" – player-controlled unit disappeared
- "newGroup" – new group with player-controlled unit
- "removeGroup" – group lost last player-controlled unit or was removed
- "newPlayer" – a new player (name) has appeared
- "changePlayer" – player has changed to a different unit (includes respawn)
- ("leavePlayer") – currently never invoked

The callback must match the profile

```
function myCallback (evType, description, info, data)
```

#### *4.2.2.2 removeMonitor(myCallback)*

Removes the previously installed myCallback from the list of callbacks that are invoked when a player event occurs.

### 4.3 cfxZones API

Trigger Zones make up the backbone of many DML modules because Zones can be easily placed with ME's graphical interface, and mission designers can easily add and modify properties to zones. Zones are ideal since they

- have a unique name and can be identified individually
- have a location on the map and therefore can pass location information that can be modified inside ME
- occupy a surface and can therefore pass area information that is easily edited in ME
- can be easily identified and modified in ME
- can pass arbitrary string data via ME-editable properties to modules that look for them

Consequently, cfxZones provides strong support for Zone management: the API provides functions for

- **Testing / Management:**  
Find zones with names or that contain part of a name, are within other zones, have proximity to locations (points or units) and vice versa (return units in a zone), or have a certain property/attribute.  
Other methods facilitate linking a zone's location to a unit, pick a zone by random, or move the zone's location.

Many of these methods allow testing against not just one, but a set of zones. Since cfxZones knows all Trigger Zones defined in ME, it defaults to testing against that set by default, making your zone management code very concise.

- **Property Management**  
Properties are the main avenue to extend Trigger Zone functionality, and cfxZones provides comprehensive support to access these properties
- **Miscellaneous**  
Since Trigger Zones are so versatile, cfxZones provides a handful of miscellaneous methods to accomplish other tasks, like placing smoke and the incredibly powerful ability of spawning units.

#### Important Note:

cfxZones copies all zones from DCS on start and creates a wrapper with enhanced data for all other modules. If you change a cfxZone, those changes will not propagate to DCS's game engine. This means that if you use cfxZones to manage your zones (especially should you change a zone's location or size), you must also use the methods provided by cfxZone's for testing to ensure that the results are correct. Furthermore, be advised that currently, cfxZones and DCS moving zones don't mix well, so choose either.

### 4.3.1 Testing

#### 4.3.1.1 *isZoneInsideZone(innerZone, outerZone)*

Returns true if innerZone's center point is inside outerZone

#### 4.3.1.2 *getZonesContainingPoint(thePoint, testZones)*

Iterates the table testZones and returns all zones from that table that contain thePoint. If testZones is nil, all cfxZones are tested.

#### 4.3.1.3 *getFirstZoneContainingPoint(thePoint, testZones)*

Iterates the table testZones to find a zone that contains thePoint, returning the first zone it finds. If testZone is nil, all cfxZones are tested.

#### 4.3.1.4 *getAllZonesInsideZone(superZone, testZones)*

Given a superZone, returns all zones contained in the table testZones that have their central point inside superZone. If testZones is nil, all cfxZones are tested against superZone. Note that even if superZone is part of testZones, it will not be included in the result.

#### 4.3.1.5 *groupsOfCoalitionPartiallyInZone(coal, theZone, categ)*

Returns a table of all all groups that match coalition coal and are of type categ and that have at least one living unit that is placed inside theZone.

#### 4.3.1.6 *isGroupPartiallyInZone(aGroup, aZone)*

Returns true if any living unit of aGroup is inside aZone

#### 4.3.1.7 *isEntireGroupInZone(aGroup, aZone)*

Returns true if all living units of aGroup are inside aZone

#### 4.3.1.8 *pointInZone(thePoint, theZone)*

Returns true if thePoint (x, y, z) resides inside theZone. Note that the y component of thePoint is ignored.

#### 4.3.1.9 *unitInZone(theUnit, theZone)*

Returns if theUnit is inside theZone

#### 4.3.1.10 *unitsInZone(theUnits, theZone)*

Returns a table that contains those units from the input table theUnits that are inside theZone

#### 4.3.1.11 *closestUnitToZoneCenter(theUnits, theZone)*

Returns the units from the input table theUnits that is closest to the main point (as placed in ME) for theZone.

## 4.3.2 Management

### 4.3.2.1 *offsetZone(theZone, dx, dz)*

Moves theZone's main point (as originally defined in ME) by dx and dz units (meters). This change will not propagate to DCS's game engine, so `trigger.misc.getZone()` for the same zone will not reflect the offset.

### 4.3.2.2 *moveZoneTo(theZone, x, z)*

Moves theZone's main point to the absolute location as defined by x and z. This change will not propagate to DCS's game engine, so `trigger.misc.getZone()` for the same zone will not reflect the move.

### 4.3.2.3 *centerZoneOnUnit(theZone, theUnit)*

Moves theZone's main point to coincide with the location that theUnit currently occupies. Note that if theUnit moves later on, theZone will **not** move with theUnit; this method does **not** link theZone to theUnit (use `linkUnitToZone()` for that, see below). This change will not propagate to DCS's game engine, so `trigger.misc.getZone()` for the same zone will not reflect this.

### 4.3.2.4 *zonesStartingWithName(prefix, searchSet)*

Returns a table of all units that start with the string prefix from the table searchSet. If no searchSet is provided, all zones currently known to cfxZones are searched.

### 4.3.2.5 *zonesStartingWith(prefixes, searchSet)*

Returns a table of all units passed in searchSet that start with any of the prefixes passed in the prefixes table. If no searchSet is given, all zones currently known are searched. If prefixes is a string, this is the same as `zonesStartingWithName()`.

### 4.3.2.6 *getZoneByName(aName, searchSet)*

Return the zone from searchSet whose name (uppercase) exactly matches aName. If no searchSet is provided, add zones known to cfxZones are searched. If no zone is found, result is nil

### 4.3.2.7 *getZonesContainingString(aString, searchSet)*

Returns a table of all zones from searchSet whose name contains aString. If no searchSet is given, all zones currently known to cfxZones are searched. If no zones are found, result is an empty table.

#### 4.3.2.8 *getZonesInRange(point, range, theZones)*

Returns a table that contains all zones from theZones whose main point lies at maximum range meters away from point. If no theZones is specified, all zones known to cfxZones are searched.

#### 4.3.2.9 *getClosestZone(point, theZones)*

Returns the zone from theZones whose main point is closest to point. If no theZones are given, all zones known to cfxZones are searched.

#### 4.3.2.10 *pickRandomZoneFrom(zones)*

Returns a random zone from the table zones. If no zones given, all zones from cfxZones are used.

#### 4.3.2.11 *linkUnitToZone(theUnit, theZone, dx, dy)*

Links theZone to theUnit. From now on, theZone's center will be placed to coincide with theUnit, offset by dx and dy (with dy being the Z offset). If dx and dy are omitted, the zone is always centered on theUnit. If theUnit is destroyed, theZone will remain at theUnit's final location (plus/minus offset).

#### **Notes:**

- The unit you link the zone to must exist, or the result is undefined.
- Zone position changes from linked units do not propagate to DCS's game engine, so `trigger.misc.getZone()` for the same zone will not reflect the link.

#### 4.3.2.12 *createSimpleZone(name, location, theRadius, addToManaged)*

Most zones managed by cfxZones are derived from the mission as designed with ME. That does not, however, mean that mission designers can't create cfxZones while the mission is in progress - it might even be required.

This method returns a new circular cfxZone with name as its name, located at location (only x and z are used, y is ignored) with the circle having a radius of theRadius (in meters). If addToManaged is true, the new zone is added to cfxZones set of managed and monitored zones: it will be updated when linked to units, and considered for all zone testing when the full cfxZones set is used (e.g. by passing nil as a test set).

Note that any zone created with this method can't be accessed via `trigger.misc.getZone()`, even if addToManaged is true; DCS game engine doesn't know about it.

### 4.3.3 Properties

#### 4.3.3.1 *getPoint(aZone)*

This is the **main accessor method to get aZone's location**. It returns a new point table (x, 0, z) that

- reflects the accurate location of the zone (in cfxZone context, including linked zones, and zones that have been moved by `offsetZone`, `moveZone` or `centerZone`)
- you can modify the contents of the returned point without worrying that the changes flow back to the zone.

#### 4.3.3.2 *getZonesWithAttributeNamed(attributeName, testZones)*

Returns a table of all zones from `testZones` that have an attribute named `attributeName`. If no `testZones` are provided, all zones known to `cfxZones` are tested.

#### 4.3.3.3 *zonesWithProperty(propertyName, searchSet)*

Alias for `getZonesWithAttributeNamed()` (see above)

#### 4.3.3.4 *getAllZoneProperties(theZone, caseInsensitive)*

Returns a table of all properties, indexed by property name, that were previously assigned to `inZone` with ME in DCS. If `caseInsensitive` is true, all property names are converted to uppercase (and might overwrite those properties that – foolishly – are only distinguished by upper/lowercase spelling).

#### 4.3.3.5 *getZoneProperty(cZone, theKey)*

Returns the value of the property named `theKey` from `cZone`. If the property does not exist for `cZone`, `nil` is returned.

#### 4.3.3.6 *getStringFromZoneProperty(theZone, theProperty, default)*

returns the string value from the property named `theProperty` from `theZone`. If `default` is not defined, `default` will use `""`. If the property does not exist for `theZone`, `default` is returned.

#### 4.3.3.7 *getMinMaxFromZoneProperty(theZone, theProperty)*

Attempts to return an array of two numbers from the property named `theProperty` of `theZone`. It assumes that the value of `theProperty` fulfills the following

- the value string begins with the first number (no leading characters or whitespace)
- the two numbers are separated by a blank, e.g. "12 34" returns [12, 34]
- the numbers can be fractions (e.g. "1.234 5")
- there are at least two numbers separated by blank
- negative numbers are allowed
- digits, signs, fraction points only (no coma separators)
- if more than two numbers are supplied (e.g. "1 2 3", only the first two are returned)
- the returned table is an array of the format {first, second}

#### 4.3.3.8 *hasProperty(theZone, theProperty)*

Returns true if theZone has a property named theProperty. Note that this method returns true as long as the property is defined for the Zone in MW, even if the value is empty.

#### 4.3.3.9 *getBoolFromZoneProperty(theZone, theProperty, defaultVal)*

Returns the value of the property named theProperty of theZone, interpreted as a Boolean. If the property is not defined, or can't be interpreted as a Boolean, defaultVal is returned. If defaultVal is not defined, `false` is used.

Note that the following values can be interpreted as Boolean

- "yes", "true", "1" are all interpreted as true
- "no", "false", "0" (Digit Zero) are all interpreted as false

Values are case insensitive, so "yes", "Yes" and "YES" will all be interpreted as true.

#### 4.3.3.10 *getCoalitionFromZoneProperty(theZone, theProperty, default)*

Interprets the value of theProperty in theZone as a coalition value and returns the appropriate ID (0 for neutral, 1 for red, 2 for blue). If the value can't be interpreted or the property doesn't exist, default is returned. If default isn't defined, 0 (neutral) is returned.

- 0, "neutral" and "all" return as 0 (zero)
- 1. "red" returns 1 (one)
- 2, "blue" returns 2 (two)

The value is case insensitive. "Blue", "blue" and "BLUE" are all interpreted as 2 (blue)

#### 4.3.3.11 *getNumberFromZoneProperty(theZone, theProperty, default)*

Interprets the value of theProperty in theZone as a number. If the value can't be interpreted or the property doesn't exist, default is returned. If default isn't defined, 0 is returned. Numbers can be negative (leading with a minus sign "-") and fractions (e.g. "3.1415")

#### 4.3.3.12 *getVectorFromZoneProperty(theZone, theProperty, minDims, defaultVal)*

Interprets the value of theProperty in theZone as a number vector (array) with at least minDims entries as follows

- the vector elements are separated by coma ",", e.g. "3.12, 4.5, 6" returns {3.12, 4.5, 6}
- each vector element is separately interpret as a number
- if an element can't be interpret as a number, defaultVal is substituted for that element
- if no defaultVal is supplied, 0 is substituted instead
- The returned array has at least minDims elements
- If theProperty contains less than minDims number of entries (separated by space), defaultVal is entered into the return array until minDims elements are reached
- If theProperty contains more than minDims elements, all entries are returned.



#### 4.3.3.13 *getSmokeColorStringFromZoneProperty(theZone, theProperty, default)*

Interprets the value of theProperty in theZone as a smoke color, returning the color as a lower-case only string, e.g. "red"

- When the color is given as a number 0..4 that is returned as the correct color string (e.g. "1" returns "red")
- The color in the value can be any mix of upper or lower case and will return a lowercase only color, e.g. "oRanGE" will return "orange"
- The returned value is a string, lower case only
- The value of default is not validated nor verified

## 4.3.4 Spawning

*4.3.4.1 createGroundUnitsInZoneForCoalition (theCoalition, groupName, theZone, theUnits, formation, heading)*

THIS METHOD SPAWNS UNITS

Returns a new group of ground units that has been spawned into the world for theCoalition. The group is named theName (if a group of the same name existed previously, that group is immediately deleted). The group's center is located at the center of theZone. The group consists of all the units as defined by theUnits (a type string array, coma separated, see → Spawning: Type String and Type String Arrays). The units are arranged as defined by formation, and the entire formation is turned towards heading.

### Notes:

- Since only coalition (not country) is specified, the units always belong to the synthetic Combined Joint Task Force of that side
- If no groupName is specified, a name is created from "G\_" plus the zone's name.
- This method only spawns ground units

## 4.3.5 Miscellaneous

*4.3.5.1 markZoneWithSmoke(theZone, dx, dz, smokeColor)*

Places a smoke at the position of theZone's center, offset by dx and dz. Smoke's color is defined by smokeColor (a number). Note that in DCS, a smoke mark will disappear after some duration (currently 5 minutes). If you need a zone/point that is permanently marked by smoke (an auto-refresh smoke, so to speak), use the provided smokeZone module.

*4.3.5.2 markZoneWithSmokePolar(theZone, radius, degrees, smokeColor)*

Places a smoke at the position of theZone's center, offset by radius and degrees. Smoke's color is defined by smokeColor (a number) Note that in DCS, a smoke mark will disappear after some duration (currently 5 minutes). If you need a zone/point that is permanently marked by smoke (an auto-refresh smoke, so to speak), use the provided smokeZone module.

cf/x Dynamic Mission Library  
for DCS

## PART IV:

# DML TUTORIAL / DEMO MISSIONS

## 5 Tutorial / Demo missions

### 5.1 Overview

DML comes with a host of demo missions crafted to demonstrate and/or highlight certain capabilities. While most of them are somewhat contrived, they are easy to understand, and most of them – unless marked **(Lua)** – do not require any Lua know-how at all.

We recommend you read below “menu” of demos first, and then pick those that interest you most.

- **Smok'em – DML intro**  
A very small, unassuming mission that contrasts DML's way of doing things against ME's old-school approach by creating smoke all over Senaki-Kolkhi
- **Object Destruct Detection**  
Shows how DML can **detect** when a **scenery (map) object is destroyed** and automatically **set a flag** that ME triggers can read. No Lua required at all.
- **ADF and NDB fun**  
**Place an NDB** on the ground, or **have it follow a unit** (e.g. a ship) with only a few clicks.
- **Artillery Zone (ME Trigger only)**  
Shows how easy it is to set up artillery **bombardment simply by placing a zone**. Then shows how that bombardment can be triggered by ME flags. No Lua required at all.
- **Spawn Zones (ideal for building training missions and lasing)**  
This is the **archetypal air-to-ground training mission**: Targets **re-spawn indefinitely**, and do not fire back. There is also a group of JTACs that **lase targets** for the pilot. No Lua required at all
- **Moving Spawners**  
A mission that demonstrates both object- and unit-spawners with a cool twist: the **spawners move**, and the units and objects drop from vehicles and form a trail behind the vehicles that drop them. No Lua required at all.
- **Helo Trooper**  
This mission demonstrates how the Helo Troops allows you to **load any infantry into a nearby player helicopter** and how to **use spawners with the 'requestable'** attribute so infantry can be 'requested' – a feature important if you are using FARPs that can be captured by the enemy (see separate demo). No Lua required
- **Helo Cargo (requires a helicopter Module)**  
A mission that demonstrates how object spawners and cargo receivers work together to **quickly create a helicopter cargo mission** with dynamic spawns. No Lua required at all

- **ME-Triggered Spawns**  
A very simple mission that demonstrates how to use **ME flags to trigger spawning** of troops and objects. No Lua at all.
- **Artillery Zone & Artillery UI**  
Shows how, by just adding a single module a **player** can **trigger artillery** bombardment, **get directions** to artillery target zones, and **mark** these **zones** with smoke – all from the communication menu. No Lua required
- **Missile Evasion (Guardian Angel)**  
Demonstrates Guardian Angel's abilities to **remove missiles just before they hit**. No Lua required.
- **Recon Mode**  
Demonstrates the abilities of the Recon Mode drop-in module, and how targets can be added to priority- or blacklist. Shows how **recon flights (AI and player-controlled)** can have significantly better spotting abilities than DCS. No Lua required
- **Owned Zones ME Integration**  
Flag bangers ahoi! This little mission demonstrates how to set up a mission with owned zones that start a whole war once the first zone is conquered. Shows how Owned Zones change ME Flags. No flying required.
- **Player Score**  
Shows how to easily add **score keeping** and units with individual score to your mission. Also demonstrates the Player Score UI module. No Lua required at all.
- **(Lua) DML Mission Template:**  
The Lua Code Skeleton – **how to structure mission code** in DCS in general and how to use DML for even better results. **Looks at creating** your own **Config Zones** in ME **and then using them** in your code. **Requires Lua skills**
- **(Lua) Landing Counter**  
A tiny, fully multi-player compatible mission that simply counts all the landings a player (not their unit – all players can change slots at any time) does. Demonstrates **how to filter world events**. **Requires Lua skills**
- **(Lua) Event Monitor**  
A **test bed for events** – to be used by Lua beginners, and everyone who has a need to analyze the sequence of events in a mission. Does not require Lua per se, but you require Lua to make any sense of it.

#### Demo Missions to come

- CSAR
- Air Traffic
- Making a skeleton CSAR mission
- Limited airframes
- jtac GUI

- making your own config zone, using libcheck
- expand on the 'writing a lua mission'

## 5.2 Smoke'em! DML Intro.miz

### 5.2.1 Demonstration Goals

This is the ideal 'Start your DML' journey, as the little 'Smoke Zone' shows us nicely how to use DML, and what's so nice about using it.



Running the demo itself isn't impressive at all. Playing with it in ME, on the other hand, is. It shows how much simpler and better even a mundane task like placing colored smoke in DML can be.

### 5.2.2 What To Explore

#### 5.2.2.1 In Mission

Run the mission and enter the Frogfoot. Then go to F2 outside view and place the camera behind the plane. On the left, there is a single red smoke. On the right, along the runway are multiple columns of differently-colored smoke. OK, so what?

Accelerate time and wait until the 5 minute mark. Aha! Not quite unexpectedly, the smoke on the left has died –this is smoke thar we created the conventional way: with a zone and SMOKE MARKER action.

#### ACTIONS

SMOKE MARKER (ME Smoke, 1, RED)

The smoke on the right, however, keeps happily on smoking. Oooh, rah, score one for DML! Yes, not that impressive, but let's move on to ME

#### 5.2.2.2 ME

##### First, The Bad

Ok, so let's acknowledge the ugly stuff first. Because it's DML, we need to load the DML modules, there's no way around that. And for that we need to have a MISSION START trigger with DCS's most intimidating action of them all: the DO SCRIPT Action *[cue scary music]!*

#### ACTIONS

```
DO SCRIPT (dcsCommon = { } )
DO SCRIPT (-- cf/x zone ma)
DO SCRIPT (cfxSmokeZone = )
```

There's no way around that wart. Luckily, it's always the same: copy/paste the entire module; usually, they are the same modules. Since DML is modular, you can often get away with only copying a few. This is DML's biggest usability issue – some people are afraid of

this first step, and it will keep them from using DML. But we are past that, intrepid mission builder, so on we go!

### Now let's try the following:

Put a red smoke marker the conventional way on the parking slots 64 – 67 (four new markers). To do this, we

- first copy/paste zone “ME Smoke” four times, and drag them to their new positions.
- Write down the four new names
- Create **four new Actions** in ME, all for the same ONCE trigger that we are using to start the one that was already there; with one of the new trigger zone names each
- And all five now die after 5 minutes

Next, try the same with DML:

- Copy/paste the zone “Smoke em!” four times, and drag them to their new positions.

With DML there are no new actions to edit, no zone names to remember - and the smoke keeps coming after five minutes.

### 5.2.3 Discussion


There's no denying it: Loading DML modules into the START MISSION trigger is ugly, frightens novices, has a decidedly 'black magic'-ish touch, and there is no easy way around it. Since it's something that you only do once per mission and then can forget about, it gets easier each time. This currently is DML's weak spot.

After that, though – using Trigger Zones and have modules attaching their magic automatically to the Trigger Zones makes editing a complex mission so much easier. You see their function on the map, and click there directly to edit.

### Still not convinced? Try this:

The conventionally – created smoke on slot 65 that you created above: change that smoke's color to white, and then the one to the right of that to blue. It's not an overly complex change, but you still need to remember zone names, open the trigger editor, got to the trigger, find the correct action, and then change the color in the pop-up.

With DML, simply click on the zone (visually identifying it **on the map**, no look-up-by-name from a list of very similar names!), and change the “smoke” attribute from “red” to “white, and to “blue” on the one to the right. How is that for quality of life?

Name	Value	
smoke	white	

Aren't you glad you went through all that DML loading trouble?

Indeed, it's still just smoke. *Unending* smoke, but still – just smoke. Try the other demos to see just how little effort it requires to add great new features to your missions. And more importantly: how easy it is to move them around and control them – right there on the map, from within ME.

Oh, and no Lua at all – let's disregard the START trigger, ok?



## 5.3 Object Destruct Detection (ME Integration).miz

### 5.3.1 Demonstration Goals

This mission demonstrates how you can integrate a DML module (the friendly object destruct detector) into your own missions, how you can trigger an action when a map object is destroyed – without any Lua code. We'll set ME's Flag 10 when our scenery object is destroyed.

### 5.3.2 What To Explore

#### 5.3.2.1 In Mission

Start the mission either as Su-27T (free with DCS) or UH-1 (not free). Then destroy the An-2M (the double-decker that is part of the scenery at Senaki-Kolkhi):



As soon as it is destroyed, you will see a message appear to that effect. Note that DML doesn't care how you destroy the Antonov. Try being creative 😊.

#### 5.3.2.2 ME

Notice the quad-based zone around the An-2. This was created with ME's "assign as" function. Click on it to reveal the zone's attributes



Notice the "f=1"/"10" attribute. That is the only addition we made to the zone. It sets flag 10 to 1 (true) when the Antonov object is destroyed (→ ME Attributes).

Now look at the ONCE trigger that runs when flag 10 becomes true. That's how easily you can integrate DML modules that can set triggers into your missions.

### 5.3.3 Discussion

This mission does not require any Lua. It uses the ObjectDestructDetector module to make testing if a map/scenery object was destroyed a snap.

Simply add a single attribute to the 'Assign as...' zone that ME created, and you are good to go. You can then use standard ME trigger conditions to test for flag change and initiate any action you like.



#### Note:

This mission may suddenly stop working correctly after an update to DCS. Sometimes, the world object DB gets updated, and object ID can get changed around. It happens rarely, and if that happens, simply right-click on the Antonov and verify that the ID's still match. If they don't update the destruct detector to new ID

## **5.4 ADF and NDB Fun.miz (tbc)**

### **5.4.1 Demonstration Goals**

### **5.4.2 What To Explore**

*5.4.2.1 In Mission*

*5.4.2.2 ME*

### **5.4.3 Discussion**

## 5.5 Artillery Zones Triggered.miz

### 5.5.1 Demonstration Goals

This mission demonstrates the use of artillery (target) zones, purely controlled via ME flags. In this mission there are artillery target zones, and multiple groups of very unlucky ground vehicles that wander into those zones.



Also, this mission shows how you can alternatively rig an 'Other...' radio item to fire into an artillery zone.

Further demonstration goals are to provide some good examples for attributes, and how you can quickly set up artillery attacks simply by placing zones.

### 5.5.2 What To Explore

#### 5.5.2.1 In Mission

Start the mission, enter your trusty 25T and observe the two 'Poor Sods' groups of ground vehicles as they wander into the "Arty Target" zone. Use outside view and F7 ground vehicle view. Cover your ears. Each time when the first vehicle enters the zone, a fire command is given to the artillery target zone (you'll see a message to that effect show up in the upper right corner). The zone then simulates a 20 second projectile transition time, after which 17 shells hit the ground in the zone.

Note that the artillery zone can be triggered multiple times

Now turn your gaze towards that group to your left. Go to communications, and choose "Other→Artillery Fire on Unlucky"

Notice that almost instantaneously, shells explode around those vehicles.

Switch to F10 map view, and note the circle mark where you just had death rain on the Unlucky. Click it to disclose the target information.

Notice how you can't trigger this explosion again via radio

### 5.5.2.2 ME

Look at the trigger Zone “Arty Target”. This is the target zone that is triggered multiple times. First, look at the attributes:

Note how we use very few actual attributes, and simply accept the factory defaults. Note the “f?” attribute that tells artillery zones to monitor flag 100 for change. Note that we override only shell count (17) and strength (700), so when the artillery zone is triggered, we expect a roughly 20 second transition time (default) for the first shells to arrive. There will be 17 shells with a power of 700 plus/minus 20% (default deviation). Note that this artillery zone does not require a collation (it defaults to neutral), and thus is not visible on our F10 chart, but will still happily respond to our commands (and destroy any troops within)

Name	Value	
artilleryTarget	One	
f?	100	
shellNum	17	
strength	700	

CONDITIONS	CLONE	^	v	ACTIONS
PART OF GROUP IN ZONE (Poor Sods, Arty Target)				FLAG INCREASE (100, 1)

So how is the artillery fire triggered? When flag 100 changes. In order to have it fire multiple times, we have rigged two triggers in ME that instead of setting flag 100 to true, **increase** the flag’s value. This change is what triggers the multiple fire cycle in the artillery zone, once each per trigger.

Now let’s look at the “Comms Arty Zone”. This one has some more attributes than Arty Target, so lets look at the differences:

This zone is triggered by flag 110 (“f?=100”), transition time is 0.1 seconds (near instantaneous) and coalition is “blue”. Also, we are to expect 22 shells with a strength of around 300 each.

Name	Value	
artilleryTarget	Two	
f?	110	
shellNum	22	
strength	300	
coalition	blue	
transitionTime	0.1	

Why the ‘coalition=blue” attribute? Recall that the F10 in-game map showed this target zone as a marker. The coalition=blue attribute allows this marker to show up on blue maps.

Recall that the shells arrived almost instantaneously. This is controlled by the transitionTime=0.1 attribute. Note that there is still some lag, and not all shells arrive at the same time. This is intentional, since not all artillery guns fire simultaneously.

So how did we wire up the artillery to fire when we command it on the radio? First, note that this zone is watching flag 110 for a change. Now look at the trigger we created

TRIGGERS	CLONE	^	v	CONDITIONS	CLONE	^	v	ACTIONS
1 ONCE (Install Comms Trigger F 110, NO EVENT)				TIME MORE (1)				RADIO ITEM ADD (Artillery Fire at Unlucky, 110, 1)

This installs a radio item “Artillery Fire at Unlucky” and when chosen, will set flag 110 to 1. This will trigger the artillery cycle the first time. If you later choose this radio item again, nothing happens because the flag value is already 1, and the artillery zone watches for a *change*.

### 5.5.3 Discussion

This mission requires no Lua at all.

#### Easy to trigger / use copy/paste

Artillery zones allow you wire up destruction all over the map in just a few seconds. The trigger watch system for artillery zones is easy to understand and works well with all standard triggers in ME, giving all mission designers easy options to command artillery fire with a radio call. Unlike many other features, it's also easy to wire artillery zones up to fire multiple times simply by changing a flag value. Also remember that multiple artillery zones can watch the same flag, making it possible to rig some spectacular firework with the same trigger simply by using copy/paste. It's also good to recall that artillery zones work over water as well as on land, and this can be used to great effect.

#### Versatility through Attributes

The way that artillery zones use attributes also makes it very easy to set up very varied explosions, and time them very precisely, simply by tweaking a few entries in ME. This makes calling in 'arty strikes' to completely destroy structures on the map a snap (remember that you can use object destruct detectors to ensure that this has really happened), and creating dramatic scripted intros (like having your airfield shelled while you take off and miraculously not damage your aircraft) easy and fun to set up. If you look at the side-by-side screenshots at the beginning of this section, you'll notice that the craters in the in-game footage matches up nicely with the artillery zone. Just remember that when you need that kind of precision, you'd need to test and probably use low-power explosions, because if you are only a few feet away from the artillery zone, just like in real life, the blast wave can kill you.

#### Further thoughts

Be advised that artillery zones function very well by themselves, and have additional built-in functionality when used with further add-ons like artillery UI (which is used to simulate FO with helicopters). We'll revisit that function soon.

And of course, as an extension to standard cfxZones, artillery zones can be linked to units – just factor in the transition time when you use such a set-up (the impact point is set at fire time, and not where the unit will be after the transition timer runs down). Used with ships and short transition time this is sure to create a great spectacle!

#### What to try

- Use an artillery zone to destroy a map object, and an object destruct detector to detect the destruction. If you can't get it to work, compare your solution with how the Artillery UI does it.
- Set up an artillery zone that follows a ship (via linked unit) and watch the great effects that you can have with that.

## 5.6 ME Triggered Spawns.miz

### 5.6.1 Demonstration Goals

This mission demonstrates how to spawn troops and objects using ME triggers. This is very useful to spawn surprises on unsuspecting pilots, or spawn cargo for transport craft.





### 5.6.2 What To Explore

#### 5.6.2.1 In Mission

Enter the cockpit of your trusty Su-25T. Your objective is to “steal” the plane. You won’t get far, though, because the moment you leave the hangar, units spawn all around you, and the taxiway to your left is suddenly blocked by three objects

#### 5.6.2.2 ME

If you have looked at the other spawn demos, there are few surprises here. The only change is the “f?” attribute for both troop and object spawners. All spawn zones (troop and object) watch the 100 ME flag for change and spawn when that flag changes value.

Name	Value	
objectSpawner	road block	
types	tetrapod_cargo	
paused	yes	
f?	100	

The 100 flag is set when the Frogfoot leaves its own “Stay inside me” zone.



As soon as the flag is changed, the spawn zones (keyed to the 100 flag by the “f?=100” attribute respond by each running through one spawn cycle.



Note that we used minimal spawners and no base name, so we could easily use copy/past for all spawners. Note in-game that the objects and units have ugly names as a result.

### 5.6.3 Discussion

This mission requires no Lua at all.

There are a couple of points to observe:

- The spawners use very few attributes and thus use default values.
- Look at the names assigned to the spawned units.  
Because the spawn zones have no '**baseName**' attribute (which is used to generate spawned unit names), all spawned objects have auto-generated names. They are ungainly and non-descriptive (e.g., "SpwnDflt-1-12"). If you turn on full labels in your mission, you may want to think about providing a more elegant base to name troops off by providing your own "baseName" attribute. Just remember that each baseName attribute must be unique per Spawner.
- All zones watch flag 100. Not a surprise, but just in case you wondered: yes, multiple zones can trigger on the same flag





## 5.7 Spawn Zones (training and lasing).miz

### 5.7.1 Demonstration Goals

This fully MP capable mission demonstrates spawn zones with different orders and respawning behaviors. It also demonstrates how to place troops that automatically lase enemy vehicles, and communication with JTACs. The spawn zones in this mission are

- *Vehicle spawn zones (targets)*  
These vehicles spawn as red. Since they have “training” orders, they won’t shoot back. Once they are all destroyed, the spawner re-spawns the entire group. The groups use different spawn formations
- *Infantry JTAC spawn (lasing)*  
An infantry group that has orders to “lase” enemy units. They lase the first living enemy unit they find that is visible (LOS) and in range.
- *Spawn anywhere demonstrator*  
An infantry group spawned on a gas platform offshore, where you can’t place them with ME



### 5.7.2 What To Explore

#### 5.7.2.1 In Mission

This is the **quintessential ground attack training mission**: unlimited weapons, unlimited targets, lasing support, no return fire – **put together in under 3 minutes!** You can add your own aircraft and they are automatically supported.

Pick an airframe (the Su-27T is free with DCS, so it is always available). Note the message from the JTAC that inform you that they have started lasing. If you fly an LGB or APKWS equipped plane, the laser code 1688.

Go to *Communications* → *Other* → *JTAC Lasing Report* to receive routing information for all currently lased targets (since we only have one JTAC group there is only one entry in the list)

Destroy the red vehicles (you have unlimited ammo). After you destroy a group of ground vehicles, they will re-spawn (after a delay of 60 seconds). Respawns are unlimited

Now destroy the JTAC group. They won’t respawn.

Finally, use F7 to cycle through all the troops until you see the group of infantry spawned on top of a gas platform.

#### 5.7.2.2 ME

There are three zones placed with ME:

- Two Spawners for BTR and Spawn Leos (defined in the types attribute). These spawn the ground troops. Since their orders are “training”, they won’t return fire. They are set to unlimited respawn (no maxSpawn attribute) after 60 seconds of cooldown. Their formations are slightly different (2 columns and rectangular)
- One spawner for JTAC (four infantry with lase orders). They have a maxSpawns of 1, meaning that they won’t respawn

Name	Value	
spawner	training Leo	
types	Leopard-2	
typeMult	9	
country	0	
baseName	Leos	
orders	training	
cooldown	60	
formation	rect	
heading	270	

Other points of interest:

- The start trigger that loads all modules also includes the jtacGrpUI module (a drop-in enhancement) so players can communicate with their JTAC
- This mission is fully multiplayer capable
- There is **no dedicated mission code**, this mission purely relies on ME Zone Enhancements and drop-in modules
- We include **no configuration zones** for any module as we like their default settings well enough

Finally, move up to the North, somewhat northwest of Sochi-Adler. See how that spawner (“Gas Platform Spawn”) is positioned on a map object. Now try to drop an Infantry group with ME there.

When you examine this spawner’s Attributes, notice how rudimentary the information is. Most spawners require only very few attributes, as they make (mostly) sane assumptions about what a mission designer would want.

### 5.7.3 Discussion

There is no dedicated mission script – this is a mission constructed entirely in ME, in very short time: import modules, place three zones, done - all modules that are include are stand-alone or work entirely with ME Attributes. In other words: this mission requires no Lua. At all.

The spawn zones for the ground vehicles provide unlimited waves of harmless targets (their orders are “training”, meaning they will not shoot back). The two spawn zones use two different spawn formation to illustrate that capability.

Since ME does not validate if a Trigger Zone is placed over land or sea, we can use this to place units in locations that are disallowed in ME. The Unis Spawner “Gas Platform Spawn” exploits this to place units on a gas platform, off the coast of Sochi-Adler. Note that this doesn’t always work (some map objects lack the required hit boxes and the troops fall

though), but this opens some interesting venues for new ideas to place troops, and even allows for some initial “liberate the pirated platform/ship” scenarios.

The JTAC spawn once and will not respawn (their maxSpawn attribute has a value of 1) and automatically lase the first enemy unit that gets in range (set to 3000 meters) and within direct LOS. Whenever they lase status changes (starting to lase, lose sight, target killed), they’ll report to their side. Note that currently JTACs always use 1688 as designation code.

Also included in the mission is the functional drop-in “jtacGrpUI” that allows all players to communicate with their JTAC to receive vectoring. A separate demo mission illustrates that in more detail.

## 5.8 Moving Spawners.miz

### 5.8.1 Demonstration Goals

This mission demonstrates how you can link spawn zones (both unit and object spawners) to other units and have the location of the spawn move. Mission designers use this to simulate both units ‘dropping cargo’ or deploying troops along a route

### 5.8.2 What To Explore

#### 5.8.2.1 In Mission

Simply sit in the cockpit and observe the two trucks on either side of the runway. The left truck drops objects (a tire with a red flag in the center) and the right truck drops infantry (a soldier). The drops occur every ten seconds (cooldown) for a total of ten times each (maxSpawns)

Not that the units/objects aren’t spawned in the center of the moving trucks, but to the side, and slightly behind.

#### 5.8.2.2 ME

There are two spawn zones: one used as a unit spawner and one as object spawner. Inspect the zones in ME, and take note of the following:

#### Unit Spawner

- **autoRemove** is set to true to immediately restart cooldown timer and respawn
- **linkedUnit** is Truck One, and **useOffset** is set to true so the ‘drop point’ for the unit always stays in the same relative position to the truck’s center.
- **formation** is set to **Line** so the single unit is always placed at the exact location of the spawn zone (a positioning feature that only the “Line” formation provides → Spawn Formations)
- **maxSpawns** is set to 10 (ten), so spawning stops after the tenth spawn

#### Object Spawner

- **linkedUnit** is the second truck (“Dropper Two”), and **useOffset** is turned on. Since this an object spawner, we don’t want the spawned objects to move with the linked truck, and therefore there is also an **autoLink = false** attribute
- **types** are two objects (Flag and Tire), and they are always created together **count** times per spawn (one) for **maxSpawns** iterations. Since count is one, the combined static object is created in the center of the spawn zone, not arrayed around the zone’s perimeter
- **autoRemove** is set to true to immediately start the cooldown and respawn cycle

### 5.8.3 Discussion

Again, this mission requires no dedicated mission script, everything works out-of-the-box – this mission requires no Lua. Even though it looks as if the truck “drop” objects and personnel, in reality this is done with spawn zones that simply use their zone’s **linkedUnit** attribute to follow a unit around.

Since object spawners are mostly linked to moving units to spawn cargo or landing markings on ships, the objects usually spawn linked to the same unit as the spawn zone itself. We need turn that off in order to have the spawned objects be 'dropped' to the ground. Although technically, DCS supports linking objects to other moving objects than ships, the results range from 'interesting' to 'unpredictable', so it is not recommended. Still, the code will not automatically unlink objects when their link type isn't a ship for future compatibility.

## 5.9 Helo Trooper.miz

### 5.9.1 Demonstration Goals

You need a troop transport helicopter for this mission: UH-1, HIND or Hip.



This mission demonstrates how players can use a troop transport helicopter to pick up infantry groups and deploy them (automatically and manually), and how to interface with spawners that use the 'requestable' attribute to selectively spawn groups.

### 5.9.2 What To Explore

#### 5.9.2.1 In Mission

Start the mission in the SU-25T. Note that there is no Other... menu available. Look around. You see that there are several groups of units already deployed on the ground, to the left (north-east):

- Pick Me Up
- Pick Me Up Too
- Illegal Team

Now start in one of the transport helicopters (for this demonstration, we'll take the Huey).

Choose Communication and observe that there is an 'Airborne Troops...' menu. **THIS IS NOT THE MENU YOU ARE LOOKING FOR!** Helo Troops provides its own menu in the "Other..." communication tree.

Choose communication→Other→Airlift Troops. This is the menu that connects to Helo Troops.

Note that the menu leads with Helo Troop's settings: Auto-Drop (currently ON) and Auto-Pick-Up (currently off). Select these to toggle their settings

Now for the more interesting stuff

### **Requesting A Spawn**

You have the option to request spawning a group Legal Team Six. If you choose this option, the spawner connected to this menu item (automatically made by Helo Troops) causes that spawner to spawn a group and then undergo cool-down. Once the group is spawned, you can pick it up like any other group. Try again to spawn the group. If you are quick enough, you'll only get a message that the spawner is cooling down (well, you get a more appropriate message, but the cooldown is triggering this message).

Using spawners with requestable spawns versus immediate spawns is useful when you want a spawner to hold back spawning until the helicopter is very close, or the spawner sits in territory that can potentially be conquered (and spawned troops will immediately start fighting)



### **Loading Troops**

Since Auto-Pick-Up is turned off, the helicopter didn't load the closest team, and you can choose which team to load. You have two options (three if you requested a spawn): Pick Me up, Pick Me Up Too (and Legal Team Six). Note that you do not have the option to load 'Illegal Team': they consist of infantry and an 'illegal' unit, the Hummer. Note also that the Team Missileer Pickup is also not available although it fully consists of 'legal' troops: it's too far away.

Choose one of the legal teams and load them up. They'll disappear from the game. Try to load another team. That's impossible, instead you have the option to disembark (deploy) the currently loaded team.

### **Deploying Troops (auto-deploy)**

Now make sure that you have selected 'Auto Drop ON'. Take off, fly to the runway's center line close to the Su-25T, and land the helicopter on the runway. Your group of infantry disembarks immediately, deploying into a defensive circle around the helicopter.

### **Auto-Pickup**

Now make sure that auto-pickup is turned on. If you do this while still on the ground, note that the troops surrounding your helicopter are not immediately loaded – auto-load only

happens the moment that your helicopter lands so you can safely change options while on the ground.

Take off empty, and touch down close to the Missileer group. That group is immediately loaded.

### **Manual Drop-Off**

Now ensure that you have set Auto-Drop to OFF. With the missileer loaded into your helicopter, fly back to the runway where you unloaded the first group. Touch down and note that your infantry stays on board. Now choose Deploy Team to have your team of missileers disembark.

### **Weight Considerations**

Currently, Helo Troops does not factor in the weight of troops it's loading into the helicopter. This is to be implemented later.

#### **5.9.2.2 ME**

Note that this mission shows a couple of important features:

- You now can pick up any friendly group that consists entirely of infantry, provided you land close enough. These units can be placed with ME or spawned from DML spawners
- Helo Troops offers per-helicopter options to auto-load and auto-unload units.
- Helo Troops determines which groups it can load. The group with the Hummer (illegal unit for helo transport) does not appear as an option
- Helo Troops only offers to load troops that are in range
- Helo-Troops automatically interacts with spawners that are in range and offer spawn on request
- Helo-Troops automatically observes a spawner's cool-down rules after requesting troops.

### **5.9.3 Discussion**

No Lua is required at all.

Helo Troops helps to integrate helicopters better into a mission – you no longer have to bother with embark/disembark waypoints: you now can pick up and deploy troops wherever you see is right.

There is more, though, so try this:

- Load up a group of infantry, and drop it close to the refueler (ground unit) at the end of the runway. The group immediately engages it until it is destroyed.





- Pick up a group of infantry, and fly due west (bearing 260). At the coast, there are a gas platform and a large tanker. Land on them. Yes, your troops can deploy on those objects! And yes, you can pick them up from them!

## 5.10Helo Cargo.miz

### 5.10.1 Demonstration Goals

This mission shows how object spawners **spawn cargo**, and how cargo **receivers** then **guide the helicopter** pilot towards the receiving zone. This mission is fully multi-player capable.

The mission dynamically spawns the cargo objects, and the receiver zone uses ME flag 10 to count the number of objects delivered. We use standard ME triggers to output a message on the first, second and third delivery. The delivery zone is marked with dynamically spawned tires arranged in a circle to mark the delivery area.

Oh, and for visual candy we also threw in a smoke zone that permanently marks the delivery zone with green smoke.

### 5.10.2 What To Explore

#### 5.10.2.1 In Mission




Fly any of the provided helicopters. Use the standard communication menu to pick up cargo. Once hooked, slowly fly the cargo towards the receiver zone that is marked with tires and green smoke. Note that during approach text messages guide you towards the zone. Unhook the cargo inside to deliver. A message will appear. Fly back to the pick-up area and notice that the cargo has re-spawned (triggered by delivering it, at which point it was deleted). Pick up another cargo and deliver, then again. Note that each time you deliver, a different message appears

#### 5.10.2.2 ME

Cargo is not placed in ME as objects, but we use object spawners set up to dynamically spawn cargo. These spawners can indefinitely supply new cargo objects. The problem with these cargo units: ME currently does not have the ability to set flags if you delivered dynamically spawned cargo, it can only work with cargo objects that exist at the start of the mission. Not so DML: we have cargo receivers that can work with any cargo.

We are **not using** any of the ME-supplied cargo triggers (CARGO UNHOOKED IN ZONE – which are woefully inadequate here because they require that the cargo is defined when the mission starts). Instead, the cargo delivery zone

uses an “**f+1**” attribute on flag **10**, which increases the flag’s value each time that you successfully deliver cargo. **The messages are triggered by using standard ME flags**

Name	Value	
cargoReceiver	can receive cargos	
autoRemove	yes	
f+1	10	

CONDITIONS	CLONE	^	v	ACTIONS	CLONE	^
FLAG EQUALS (10, 2)				MESSAGE TO ALL (Second Cargo De, 30, false, 0)		

Note that we use a separate object spawn zone to create the ring of tires that marks the receiver zone. We could have “stacked” the zones by using only one zone and move all attributes into one zone, but for clarity (and a possible attribute conflict) we use separate zones to separate cargo receiver and object spawner.

Also note the permanent smoke that is positioned slightly off the cargo zone. It only adds some visual pizzazz, and nicely shows how to add an 'eternal' smoke marker.

### 5.10.3 Discussion

This mission requires no Lua at all.

Cargo delivery Zones allow you to work with dynamically spawned cargo – something that ME currently unfortunately doesn't allow at all. So whenever you are designing a mission where cargo can appear as a result of mission events, cargo receivers allow you to automate hauling that cargo to the destination.

Note also that cargo delivery zones 'talk to pilots' to guide their cargo, a great help for the final meters during delivery. This ability is built into cargo delivery zones, the messages only appear to the helicopter group hauling the cargo, and the directions only commence on the last few meters.

#### How Cargo Spawner, Cargo Manager and Cargo Receiver interact

Here is how Object Spawn Zones, Cargo Manager and Cargo Receiver Zones work together:

- At start, the Cargo Receiver Zone requests that it is updated on all cargo events by registering a callback to Cargo Manager
- The object spawn zone spawns a cargo object and places it at its center. This spawn is counted against the zone's maximum number of spawns
- The Object Spawn Zone checks maxSpawns, and sees that it can re-spawn because maxSpawns are unlimited (-1)
- Since autoRemove is false (by default), the spawner watches the object and waits for the cargo to disappear before the next spawn cycle is started
- Since the managed attribute is true (by default) and the CargoManager module is loaded in this mission, the Object Spawner passes the new cargo object to Cargo Manager
- The cargo is now available in-game like normal cargo placed in ME
- A helicopter hooks, and then lifts the cargo. Note that this does *not* make the cargo disappear from the spawner's perspective. No new spawn cycle is initiated.
- Cargo Manager notices that the cargo was lifted. It notes this cargo's status as 'lifted' and invokes all subscribers with the 'lifted' event
- Cargo Receiver's callback is invoked with 'lifted'. Since the Receiver looks only for 'grounded'-events, it ignores this event.
- Once every second, Cargo Receiver Zone enquires from Cargo Manager all cargos that are currently in the air (lifted). For each one, it checks if the cargo's current position is close enough to a receiver zone for directions. If so, it checks the 'silent' attribute for that zone, and if not set, it outputs directions for the helicopter that is closest to the cargo (this is usually the one hauling the cargo, but in rare cases the wrong helicopter can receive the directions. Can't be helped)
- When the helicopter puts down the cargo, cargo manager notices and invokes all subscribers with a 'grounded' event
- The Cargo Receiver's callback is invoked with 'grounded' event. It checks the location of the grounded cargo against all receiver zones. If that location is inside a receiver zone,
  - it invokes its own callbacks with the event 'deliver'

- processes the flag information attributes. In our example, it increases the value of Flag 10.
  - Since autoRemove is true, the cargo object is deleted. This will cause the spawner to initiate the next spawn cycle (see below)
- The Spawn Zone detects that its watched cargo has disappeared from the game, and starts the next spawn cycle by cooling down for 60 seconds (default) and then spawning a new cargo object

## 5.11 Artillery with UI.miz

### 5.11.1 Demonstration Goals

ArtilleryUI is a drop-in module to control/trigger firing a firing cycle into a cfxArtilleryZone for a unit working as an Artillery Forward Observer (FO). This mission demonstrates multiple points:

- How easy it is to integrate a feature enhancement (Artillery UI)
- How Artillery UI works in missions
- How artillery zones can be used to destroy map objects
- How to use a config zone to change some behavior (e.g. smoke color)
- Use an object destruct detector to trigger an ME action when the artillery destroys a map object
- (This mission also shows how we can remove an artillery zone after the work is done using a single Lua command. Ignore this bit until you feel comfortable looking at Lua code)

Normally, Artillery UI only works with helicopters – this restriction can be lifted with an attribute in a config zone (naturally). This allows us to use the free SU-25T module as FO. Since an FO must remain in close proximity to their target zone, we use a trick and enabling active pause so the Frogfoot can function as magical helicopter.

With the Su25T fixed in place, we then demonstrate the various options that ArtilleryUI offers. There are two artillery target zones on the map: one immediately to the left of the plane, and one more than 200km to the east, in Tbilisi.

### 5.11.2 What To Explore

#### 5.11.2.1 In Mission

Start the mission and do not touch the Frogfoot's controls until the active pause kicks in. then look to the left. There is a factory complex that is one of our target zones.



Our goal is to have artillery destroy this complex. Since we are in active pause, we can take all the time in the world to experiment with ArtilleryUI.

Choose Communication→Other→Forward Observer

This is the Artillery UI interface. You have three options:

- List Artillery Targets
- Artillery Fire Control
- Mark Artillery Target

## Listing Targets

When you choose List Artillery Targets, artillery UI lists all currently artillery zones that cfxArtilleryZones is managing. When you are close enough to observe, your status is either listed as “OBSERVING” or “OBSCURED”

If you aren’t close enough to observe, the target zone is listed with range and bearing

Bringing down the house - OBSERVING  
Soganlug Airfield [266.8km at 88°]

In this mission we have two target zones. We are observing one (“Bringing down the house”), and the other (“Soganlug Airfield”) is 267km at bearing 88°

## Marking Targets

It’s not always (well, really never) easy to immediately spot your objective, especially if the target zone is swamped with enemies that have weapons and shoot at you – which they will. Therefore, when you are close enough (within 30km of the target zone), you can request to mark the target zone with smoke. So this now. Notice how even though there are two artillery zones on the map, you are only close enough to one, and therefore you only have one choice. Select <Bringing down the house>

Artillery shoots a single smoke round into the artillery zone, and a few seconds later, orange smoke will erupt from somewhere close to the building



Note that you do not need to be close enough to observe to have the target marked.

## Fire! Command

When you are close enough and OBSERVING (meaning that in addition to be close enough, you also have clear LOS to the target zone’s center) you can instruct artillery to fire. Doing so trigger’s the artillery zones fire cycle and then initiates a cooldown phase (artillery is reloading)

Similar to the Mark Zone command, the fire command only lists artillery zones that are available to receive a fire command, i.e. those that your unit is observing. Since your Frogfoot is hovering close enough with clear LOS to <Bringing down the house>, order the artillery to fire, and enjoy the show. Notice how the factory is levelled and you receive a message about the success.



GOOD SHELLS!

Factory is down.

Now try to issue another fire command. You'll notice that you get a 'No unobscured target areas' message. That is because the objective was destroyed, and the target zone was removed. The other target zone, Soganlug, is too far away for us, so we are done here.

#### 5.11.2.2 ME

There are two artillery zones on the map: one close by to out Su-25T, and one far away in Tbilisi. Inspect the attributes in the artillery target zone, and note

- **coalition** is set to blue. This is important so artillery UI shows this target zone to blue side
- **transitionTime** is set to 5 seconds. This is just to make us wait less time. Note that transition time affects both the smoke petard and artillery shells

There are a couple more items that are noteworthy:

- There is a **ArtilleryUIConfig** zone. This configures ArtilleryUI so that aircraft can also use the UI, and sets the smoke color to orange (it's red by default)
- There is a strange second ME Trigger Zone inside the artillery zone: "ceh\_ang\_b". Inspect it and you will find that this is a Trigger Zone created in ME with "Assign as", and is used as an **Object Destruction Detector!**
- The Object Destruct Detector increases flag 100 (Attribute **f+1=100**) when the building is destroyed.
- So what does flag 100 control?  
Inspecting the trigger in ME reveals that it does a couple of things: it outputs the "Good Shells" message. This is how your mission can use object destruct detectors to control other aspects of your mission and trigger actions
- **(Lua Only)** There is another action triggered with flag 100: a DOSCRIPT action. You should ignore it for now, but what it does is remove the artillery zone from cfxArtilleryZones list of managed zones to it disappears from artillery UI. This is merely eye candy, and when you feel ready to jump into the Lua abyss, you'll find that this wasn't actually that difficult to find out (simply look up cfxArtilleryZones API and find out what `removeArtilleryZone(zoneName)` does



#### 5.11.3 Discussion

This mission requires (almost) no Lua at all.

As you can see, merely adding the Artillery UI to the mission gives you access to an entire UI for helicopters to mark artillery zones and FO visibility logic.

We used a config zone to change the way ArtilleryUI normally works in two ways:

- The UI is also available to fixed-wing aircraft (instead of helicopter only)
- Smoke color to mark the target zone is set to orange instead of default (red)

We used an Object Destruct Detector to find out when the factory is destroyed and used that to trigger an action (a message to everyone)

This mission also uses a tiny bit of black Lua magic to remove the target zone from the pool of managed target zones after the objective was achieved (we detected that the map object – the factory). It does not affect how the mission works, just adds some polish.

**What to try**

Use ME to change the configuration zone and add attributes for allSeeing, allRanging and allTiming and then see how this affects your ability to trigger the Soganlug artillery zone. Use F7 to observe the bombardment (there is a vehicle “Kenny” there).



## 5.12 Missile Evasion (Guardian Angel).miz

### 5.12.1 Demonstration Goals

This demonstrates the drop-in module “Guardian Angel”, a module that protects all player aircraft from missile attack. In this demo, we turned on the showy (and potentially harmful) ‘explosion’ effect that “detonates” missiles instead of removing them.

It also shows how AI planes can be added to Guardian Angel’s watchlist (Lua only)

### 5.12.2 What To Explore

#### 5.12.2.1 In Mission

Fly the Frogfoot along the route, and keep around 2000m altitude. Notice the frightening Hydras of missile contrails building as missile after missile is launched from multiple SA-6, S-10 and S-11 sites.



Do not try to evade. Note that after a short while, all planes except yours and a Jeff are dead. Note the warnings and other messages on the right side of the screen. Note that even if you don’t try to evade or expend any counter measures, you still live through the flight.

Also note that missiles that are removed by interventions explode at a safe distance

#### 5.12.2.2 ME

Note the configuration zone. If you inspect it, you will see that we enabled explosions for effects, and set the value to 1.0. Note that this can potentially harm other aircraft.

Inspect the triggers and note the ONCE (Protect Jeff One) trigger. This is a bit of Lua code. It shows how, when you know the name of an AI Unit, you can also add it to the list of protected planes.

### 5.12.3 Discussion

This mission requires no Lua at all.

Guardian Angel does its job really well, allowing missiles to come close, but not too close to protected planes. You can use this for many purposes: missile evasion school (where every time Guardian Angel intervenes, you would have lost), for adding harmless but blood pressure rising drama to a mission sequence (where a player plane receives protection to ensure nothing happens), or to kick up your missile training difficulty by disabling some Guardian Angel capabilities (for example disable interventions, but keep missile warnings in place).

We have turned on the explosions effect in the configuration zone. Turn it off, and explore some other values.

### **What to try**

- Turn off explosions
- Turn on 'private' – this reduces message clutter
- Turn off intervention and see how long you can survive. Mind the Missile missile missile! warning.

## 5.13 Recon Mode.miz

### 5.13.1 Demonstration Goals

This mission shows the basic functionality of how recon planes can be used, how to add priority targets, and how to add black-listed (invisible to recon) targets.

### 5.13.2 What To Explore

#### 5.13.2.1 In Mission

Start the mission in the Su-25T on the ground. Switch to F10 Map view and simply wait while observing the Tomcat on its way in-and and (a little while later) the Albatross after it took off.

Note the circles appearing on the Map. Click one of them



Note that for most of the discovered ground units there are no DCS-provided markers on the map.

Note the text messages appearing in the upper right corner.

Note the “GOTCHA” message that appears after a minute or so. This is a **ME-triggered** message that is displayed after Recon Mode found a group that was on a priority list of targets

Note that there never is a message that a group named “never find me” appears. This is important because this is a group that does exist, in the path of the recon plane, but was black-listed, so it should not be discovered.

Now re-start the mission, and take off, cruelly ignoring the albatross. Fly into the general region where the Albatross discovered the ground units. Note that your plane also automatically reports any units found.

#### 5.13.2.2 ME

There are a couple of interesting points here:

- Note the red ground units as they are on the map. Notice that there are two groups of special interest to us: “Me B priority!” and “never find me”. There is nothing special about their set-up (these are standard ME-placed units) except we need to remember their name
- There is a config zone on the map that sets up two flags that Recon Mode modifies when a recon plane discovers ground forces: 100 (for normal discoveries) and 110 (for priority target discovery)
- Inspect the “Prio Detected” ME trigger. This fires when Flag 110 is greater than 0. This is how you can detect in your mission when a scout detects a priority target

- Inspect the “Six detections” ME trigger. This fires when Flag 100 is greater than 5, meaning that Recon Mode has at that point discovered six ground groups (not counting any priority group).
- **(Lua Only)** Note the “Demo: add priority/ignore groups” ME trigger. This demonstrates how you can add the name of a group to the priority- and blklist (we noted down the names above). Unfortunately, doing this requires that you understand how to write a single Lua command, so return to this when you need this feature and are ready to face this task.
- Note the lone red Albatross inbound to Gudauta. It is only included to demonstrate (when you turn on verbose in the config zone) that red planes are not added to the scout list because redScouts is turned off

### 5.13.3 Discussion

This mission requires (almost) no Lua at all.

We can add full recon flight abilities to a mission simply by adding this module.



Note how **discovered groups are marked** on the F10 map but the **red units do not show up** as symbols. This means that DCS's For of War mechanics still hide the units, making the recon flight a very useful addition for missions that center around looking for specific enemy troops.

#### What to try

- Experiment with the `announcer` and `applyMarks` attributes in the config zone to see how you can change Recon Modes messaging behavior
- add `detectionMinRange` and `detectionMaxRange` attributes to the config zone, and experiment with them. You can, for example, make your planes hyper observant by setting both values to 100000 and then watch in awe as the Tomcat and friends detect all enemy ground units within some 20 seconds.
- Create a mission with lots of ground units and many planes, and allow all planes to auto-recon. Notice that there may be a few seconds between detection of ground units now as Recon Mode minimizes performance impact (which now is next to negligible)

#### Restrictions

When incrementing ME flags, Recon Mode currently lumps detection events for red and blue together. This will be extended with new attributes in a later version. If you need more information about what side found a group, you need to use callbacks.

## 5.14 Owned Zones ME Integration.miz

### 5.14.1 Demonstration Goals

This mission shows how Owned Zones work in general and how they can be used to set ME Flags. It also offers a nice test bed to illustrate how the various cooldowns work and can influence the game.

### 5.14.2 What To Explore

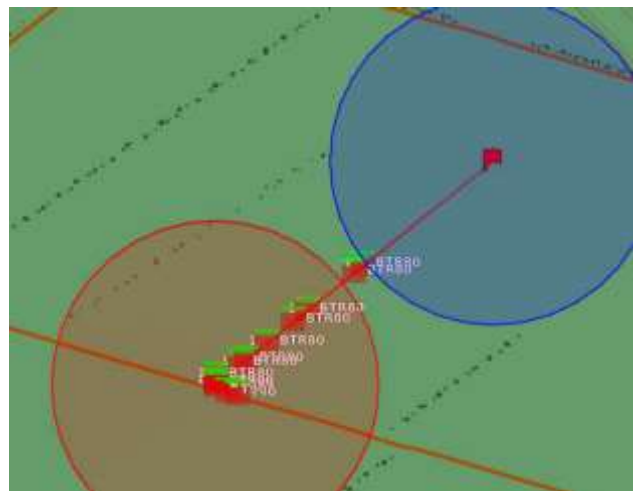
#### 5.14.2.1 In Mission

Start the mission, enter your Frogfoot, and switch to the F10 in-game map. To the south of the airfield are two blue circles. These are two zones owned by blue coalition. The southern of these has an Infantry unit that defends the zone, the northern blue zone is undefended.

A group of two red T-90 are approaching southern blue. After a (very) short battle, red wins. Upon entry of southern blue, that zone turns red, and we receive two messages: “REDFORCE have secured zone Blue Owned One” and “We won a zone”

After a short while, a red infantry appears in the newly captured red zone, and a few seconds later BTR-80s start to appear in regular intervals that move towards northern blue.

Eventually, the first BTR-80 arrive inside northern blue, and that zone is captured. From that moment on, the zone that red captured first stops producing new BTR-80



#### 5.14.2.2 ME

Note that there are no blue units on the map placed with ME. All blue troops are spawned dynamically from OwnedZone. The blue Infantry unit M4 is the blue defender that we specified in the owned zones defenderBLUE attribute: “Soldier M4”. Defenders are spawned at mission start.

Name	Value	
owner	blue	
defendersBLUE	Soldier M4	
spawnRadius	5	
attackersRED	BTR-80, BTR-80	
defendersRED	Soldier M4	
attackRadius	10	

After capture by red we produce the same unit, this time as defendersRED. Note that owned zones can spawn usually ‘blue’ units as red. Later, the BTR-80s are the units that we defined under attackersRED. Note that there are no attackersBLUE defined: this zone does not produce attacking units.

Note the two triggers “ONCE Got One Red” and “ONCE Got 2 Reds”). They both trigger on Flag 10, one when Flag 10 has a value of 1, the other when the value of Flag 10 is 2. If you inspect the ownedZonesConfig, you will find that we are banging Flag 10 for red: f!=10. Each

```
1 ONCE (Got One Red, NO EVENT)
1 ONCE (Got 2 Reds, NO EVENT)
```

time red captures a zone, this flag's value increases. Each time red loses a zone, this flag's value decreases. We trigger our message "We won a zone" on flag 10.

Owned Zones generates the "REDFORCE have secured zone Blue Owned One" message which we can turn off with the 'announcer' attribute in the config zone.

### 5.14.3 Discussion

Again, this mission required no Lua at all and integrates with normal ME flags.

Here are some other points worth mentioning and exploring

- Zones can be owned and undefended: northern blue starts as a blue zone, yet it is entirely undefended. When undefended, Owned Zones remain the possession of a coalition until the opposing coalition places at least one ground unit inside the zone.
- When a zone is conquered, a small bug in DCS may prevent it to correctly change color. Zooming in or out of the F10 map resolves that
- Like all dynamic spawners, Owned Zones can spawn lots of units in a very short time. Be careful with the `attackingTime` attribute (config zone), as that controls an Owned Zone's spawn interval. We set it to a very short interval (15 seconds between spawns) for this demo. In a real mission, spawning units every 15 seconds will create a vast number of units that quickly overwhelm the computer.
- Banging flags is a powerful feature to take advantage of: it's easy to define a win condition that merely triggers on total owned zones captured – if a side loses a zone, that flag's value decreases automatically, if they capture one, it goes up
- Be mindful of some of ME's restrictions when setting up triggers that use banged flags. Remember that ME can't compare flags to negative values (DML knows no such restrictions)
- Add a "Leopard-2" as `attackersBLUE` attribute to **northern** "Blue Owned Two". Try to predict what will happen, then run the mission. Surprised? A remarkably interesting dynamic is that Blue Owned Two reacts only *after* Blue Owned One is captured.

## 5.15 Keeping Score.miz

### 5.15.1 Demonstration Goals

This mission shows the Player Score and Player Score UI modules in action. It also demurely demonstrates a permanent smoke zone, just because we can. This mission provides unlimited ammo and targets, so you can go nuts. Targets won't shoot back.

### 5.15.2 What To Explore

#### 5.15.2.1 In Mission

Start the mission and use the (free) Su-25T or one of the A-10 (A or C) to lay waste to the poor targets on the ground. If you fly the C-Hog, there's also target lasing available with a code of 1688.

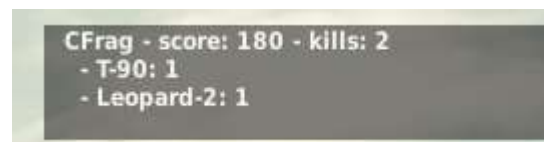
When you kill a ground unit on the main runway, note how your score increases each time the unit is killed (if the units is still "cooking off", the score is awarded only after the unit explodes). Hitting a BTR-80 yields 10 points, while a Leo nets 30.

Close to the main tarmac, marked by red smoke, are three T-90 tanks. Kill them all, and watch the score. After killing one of them you get a message that you killed a strategic target ("Big Kahuna") and receive a significantly higher score (150). Note that scores are totaled as well.



After successfully killing some vehicles (at least one BTR-80 or Leopard, and one T-90), choose Communication→Other... →Score/Kills. This is the Player Score UI module that allows a player access to more detailed personal score information. You are presented with your personal kill log:



- Total score and number of kills
- List of all types killed and their number
- Note that named kills also appear as a separate type



#### 5.15.2.2 ME

Note that there are two zones on the map that feed data to Player Score. One is the 'normal' configuration zone ("playerScoreConfig"). The other

("playerScoreTable") is much more interesting. It holds the score table for this mission. As you can see, the Unit Named "Big Kahuna" yields a score of 150 points. This is a "named unit score", as only units that match that name receive this score, and since unit names must be unique in DCS, there can only ever be one unit that is awarded that score.

Name	Value	
Big Kahuna	150	
Leopard-2	30	

Also, all units of type “Leopard-2” receive 30 points. This is a “unit type score”, because “Leopard-2” is a known type string for units of that Type. All units that match that type yield a score of 30.

Since the BTR-80 and T-90 are listed nowhere on the Player Score Table, they award only 10 points each since that is the default score for ground vehicles.

### **5.15.3 Discussion**

This mission requires no Lua at all.

Things to explore

- Change aircraft after killing some units. See that your previous score is brought over
- Play in Multi-Player. See that the score is attributed individually.
- Note the permanent smoke (red) zone that we added to better find the priority (Big Kahuna) target.
- Change the default score for ground vehicles to 25 in the player Score Config zone, and try again.



## 5.16 DML Mission Template.miz – (Lua Only)

### 5.16.1 Demonstration Goals

This mission demonstrates the following:

- Adding DML modules to a mission during Start
- Invoking DML foundation from script
- Minimal DML-based designer-authored mission script “dmlMain” that
  - Validates DML libraries
  - Reads a config zone with attributes
  - Subscribes to all DCS world events and writes them to screen as they happen
  - Subscribes to all DML player events and writes them to screen as they happen

### 5.16.2 What to explore

#### 5.16.2.1 In-Mission

Start the mission as single or multi-player. You can choose one of multiple Su-25T planes. Before you choose a plane, however, note the lines of text on the right side. They tell you that some DML modules have loaded.

More importantly, though, they also show what values the dmlMain mission script read from the configuration zone that was placed with ME.

Choose a slot, and start the mission

As soon as you enter the cockpit, new lines of text appear on the right side. These chronicle world and player events as they happen.

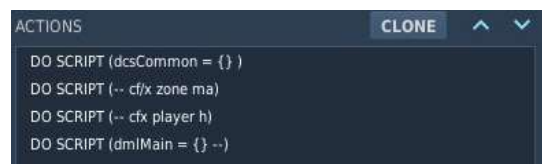
Keep an eye on the right side and now change plane slots, eject, crash the plane, and watch other planes start-up etc. As time progresses, more events are chronicled.

While unimpressive at first, simply remember that these events are what we will use to control our missions – in a much more fine-grained way than the out-of-the-box trigger conditions provided by ME can.

#### 5.16.2.2 ME

Inspect the following:

- Triggers Panel  
Inspect the MISSION START trigger. It contains a number of actions: they all are DOSCRIPT actions.
  - All except the last load DML modules: dcsCommon, cfxZones, cfxPlayer
  - The last DOSCRIPT defines the dmlMain source (dissected in Discussion, below) which is the actual mission-designer authored script that uses DML methods to perform various tasks. dmlMain runs in the background and checks the world once a second, while it is directly invoked whenever a world



```
ACTIONS
DO SCRIPT (dcsCommon = { } )
DO SCRIPT (-- cfx zone ma)
DO SCRIPT (-- cfx player h)
DO SCRIPT (dmlMain = { } --)
```

event or player event happens.

Copy the entire contents of the DOSCRIPT action into a text editor so we can discuss it later on

- Inspect the **yellow trigger Zone** named **“dmIMainConfig”** placed on the map near Senaki-Kolkhi. Notice the tribute list (Name/Value).

There are two name/value pairs defined:

- “test one” has “any value” as value
- “another test” has “42” as value



Name	Value	
test one	any value	
another test	42	
		 
		

- Run the mission again. When the mission starts and shows the available slots, note again the text on the right. Now notice that they show the same text for name/value. This demonstrates the dmIMain can easily read attributes placed in a zone.
- In ME, change the values, names, and perhaps remove and add attributes, then run the mission again. See how the changes you apply to the attributes in dmIMainConfig are reflected when the mission runs.
- Note that you can add your own aircraft and they are all automatically supported

### 5.16.3 Discussion

dmIMain follows DML’s main mission script design philosophy. It provides a very flexible and extensible bed for complex missions or designs that can’t be easily accomplished with ME.

#### IMPORTANT

Remember that even though mission scripting in Lua is by orders of magnitudes more powerful than what can be achieved with classic trigger scripting, don’t overlook the fact that one does not preclude the other. ME does provide a lot, and many missions can profit from a hybrid approach: design the easy/ornamental stuff in ME with triggers, and just do the complex stuff with mission scripts. Look how DML encapsulates functions into modules, and then uses ME Zones as interface. Whenever possible, you should emulate this concept.

That being said, let’s dig into the DML-based main mission. As stated before, with just a few lines of code, it provides almost everything you need to write exceedingly complex mission, and easily perform tricks that you can’t do in ME alone.

The basis of this mission scrip templates are

- A config zone to provide easily modifiable (In ME) mission settings
- An update loop that is invoked once per second

- Event Handlers for world- and player events. These get invoked every time something potentially interesting happens, and let dmlMain sleep otherwise.
- A start method that gets everything rolling

In other words: this dmlMain fully implements what we discussed in → DML Mission Design Philosophy.

So, let's look more closely. If you haven't done already, copy the DOSCIPT text into a text editor, and unship your looking glass – we are going in!

#### 5.16.3.1 Reading Configuration Data

Since a basic script like this does not need complex configuration data, we simply, just for fun, place a zone in ME (called “dmlMainConfig”) and add a couple of attributes. Our readConfiguration() method simply uses cfxZones to retrieve the zone with getZoneByName(), and retrieve all properties from that zone as a table with getAllZoneProperties(). After that we simply write all name/types to outText()

```
function dmlMain.readConfiguration()
    local theZone = cfxZones.getZoneByName("dmlMainConfig")
    if not theZone then return end
    dmlMain.config = cfxZones.getAllZoneProperties(theZone)
    -- demo: dump all name/value pairs returned
    trigger.action.outText("DML config read:", 30)
    for name, value in pairs(dmlMain.config) do
        trigger.action.outText(name .. ":" .. value, 30)
    end
    trigger.action.outText("---- (end of list)", 30)
end
```

#### 5.16.3.2 Main Update Loop

This simple mission script does not watch game states, and can happily live without an update loop. Accordingly nothing is being done inside the loop. Should you place a trigger.action.outText() method inside update(), you'll see a new text line every second – because that is when update is invoked.

```
function dmlMain.update()
    -- schedule myself in 1/ups seconds
    timer.scheduleFunction(dmlMain.update, {}, timer.getTime() +
1/dmlMain.ups)
    -- perform any regular checks here in your main loop
end
```

The main trick here is that update() simply schedules itself again in one second, and thus continues on indefinitely.

Yes, some things really are as simple as that.

#### 5.16.3.3 World Event Handler

We only provide two callbacks here: the pre-processor wPreProc() (which does nothing except returning true to all events are passed to the main event processor), and the main

event processor `worldEventHandler()` which merely uses `dcsCommon` to to translate the event ID to some human-readable format, and then submits it to `outText()`

```
function dmlMain.wPreProc(event)
    return true -- true means invoke worldEventHanlder()
    -- filter here and return false if the event is to be ignored
end

function dmlMain.worldEventHandler(event)
    -- now analyse table <event> and do stuff
    trigger.action.outText("DCS World Event " .. event.id .. " ("
.. dcsCommon.event2text(event.id) .. ") received", 30)
end
```

Mission code usually uses these event handlers to determine if the game needs to change state. Since our mission doesn't use states, we simply demonstrate how they are invoked and provide some in-mission feedback when they are.

#### 5.16.3.4 Start()

Quite unsurprisingly, `start()` merely connects the dots, and starts `update()`.

```
function dmlMain.start()
    -- ensure that all modules have loaded
    if not dcsCommon.libCheck("DML Main",
        dmlMain.requiredLibs) then
        return false
    end

    -- read any configuration values
    dmlMain.readConfiguration()

    -- subscribe to world events
    dcsCommon.addEventHandler(dmlMain.worldEventHandler,
        dmlMain.wPreProc) -- no post nor rejected

    -- subscribe to player events
    cfxPlayer.addMonitor(dmlMain.playerEventHandler)

    -- start the event loop. it will sustain itself
    dmlMain.update()

    -- say hi!
    trigger.action.outText("DML Main mission running!", 30)
    return true
end
```

No surprises there: integrity check (`libCheck()`), config data loaded (`readConfiguration()`), world event subscribed to (`addEventHandler()`), player events subscribed to (`addMonitor()`), and `update()` started.

#### 5.16.3.5 *Player Event Handler*

Again not really required for this particular mission, we merely demonstrate how it is used and when by writing out some text. When you run the mission you see when and with which events it is invoked. Tip: fly a helicopter (Ka-50), eject, and see the sequence of event unfold: player events and worlds events.

```
function dmlMain.playerEventHandler (evType, description, info,
data)
    trigger.action.outText("DML Player Event " .. evType .. "
received", 30)
end
```

## 5.17 Landing Counter.miz – (Lua Only)

### 5.17.1 Demonstration Goals

Shows how intercept specific world events and count all landings a player makes, independent of which plane they fly. Report, but don't count AI landings.

### 5.17.2 What To Explore

#### 5.17.2.1 In Mission

Fly the mission, and put down a couple of landings. Change the airframe after a landing, or take off, and land again. Run the same mission in Multi-Player and have other people land. See how only player landings are counted in total (not by airframe/slot they occupy), and that AI landings (that happen at certain times) are reported, but not counted.

#### 5.17.2.2 ME

Copy the source for `ldgCtr` into a text editor for discussion later. Note that there is no configuration zone anywhere in the mission.

Note that you can add your own aircraft and they are all automatically supported

### 5.17.3 Discussion

#### 5.17.3.1 Summary

Basis for this mission is that we process all world landing events, and discard all others. If a landing event occurs, we report that fact, and then see if that unit is piloted by a player. If so, we increase the number of landings for that player.

#### 5.17.3.2 Interesting Details

A couple of interesting things:

- Although there is no “`ldgCtrConfig`” zone placed with ME, we are still looking for it. This is just to demonstrate that we can safely future-proof the mission by including the code without penalty
- We don't need `update()`. Like above, we still include it
- We do not need player events. We still include the code for possible later expansion
- We have a table `ldgCtr.landings` that will contain a number for each player to count the landings
- We use `wPreProc` to filter out all events that are not landing (4)
- In the `worldEventHandler`, we know that `wPrePro` makes sure that we are only invoked for landing events, and directly process the event table
- How do we know that the plane is piloted by a player? When `getPlayerName()` returns anything but `nil`.
- We index the landing count by the player name and so can tabulate all landings by player, irrespective of the unit they are flying
- This code works with any number of players automatically, no special provisions to support multi-player required.
- You can add your own aircraft and they are all automatically supported

### 5.17.3.3 Relevant Code

So, let's look at the code, which you can extract by copy/pasting everything from the DOSCRIPT action for `ldgCtr`

```
function ldgCtr.wPreProc(event)
    return event.id == 4 -- look only for 'landing event'
end

function ldgCtr.worldEventHandler(event)
    -- wPreProc filters all events EXCEPT landing
    local theUnit = event.initiator
    local uName = theUnit.getName()
    local playerName = theUnit:getPlayerName()
    trigger.action.outText(uName .. " has landed.", 30)
    if playerName then
        -- if a player landed, count their landing
        local numLandings = ldgCtr.landings[playerName]
        if not numLandings then numLandings = 0 end
        numLandings = numLandings + 1
        ldgCtr.landings[playerName] = numLandings
        trigger.action.outText("Player " .. playerName .. "
completed ".. numLandings .." landings.", 30)
    end
end
```

### 5.17.3.4 Further Notes

Also, the `start()` function contains these interesting lines:

```
-- init variables & state
ldgCtr.landings = {}
```

Above simply initializes the landings counter to zero for all players (actually, it removes all numbers from the table), and makes sure that the table exists in the mission space so we do not crash when we try to access it from `worldEventHandler`.

#### Note

It's best practice to define and initialize module variables such as this at the very beginning, not just in `start()`. Re-initializing variables in `start()` is also good practice as it also future-proofs your mission code.

## 5.18 Event Monitor.miz

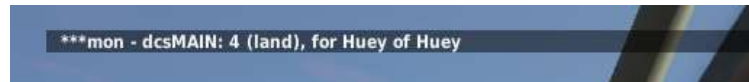
### 5.18.1 Demonstration Goals

Event monitor is a development tool mainly for mission designers who are learning Lua and advanced mission designers who need to investigate events as they happen during a mission. Event Monitor uses cfxMon to log all events as they happen to the screen.

### 5.18.2 What To Explore

#### 5.18.2.1 In Mission

Start the mission and, jump into your favorite plane and cause the events that you are looking for.



#### 5.18.2.2 ME

Note that Event Monitor is a bare bones tool template. Use it to create your own specific event tester. Out of the box, this mission only monitors the basic DCS events, and only comes with the Foundation Modules dcsCommon and cfxZones. Should you want to test your own scenarios or events, make sure to include the relevant modules (even your own).

### 5.18.3 Discussion

This mission's config zone sets the 'delay' for events to 5 seconds (default is 30). If you are new to DCS events, be sure to do the following, and note the event that creates:

- Enter a plane
- Start up a plane
- Extend and retract flaps and manipulate some other cockpit instruments (note: no event!)
- Take off in a plane
- Land a plane
- Change a plane
- Eject from plane. Let the ejected pilot reach the ground. Wait for the plane crash event.
- Crash into ground
- Fire missiles
- Fire cannons
- Drop Bombs
- Kill a target on the ground in Kutaisi
- Land a plane outside of an airport
- Take off and land a helicopter
- Crash a helicopter



## **5.19 Mission.miz**

### **5.19.1 Demonstration Goals**

### **5.19.2 What To Explore**

*5.19.2.1 In Mission*

*5.19.2.2 ME*

### **5.19.3 Discussion**

## **5.20 Mission.miz**

### **5.20.1 Demonstration Goals**

### **5.20.2 What To Explore**

*5.20.2.1 In Mission*

*5.20.2.2 ME*

### **5.20.3 Discussion**