cf/x

Dynamic Mission Library for DCS

© 2022 by Christian Franz and cf/x

Version 0.9986 – 2022-04-28

Table of Contents

1 Welcome – What is this? 11
1.1 About this Document 15
1.2 Part IV - Demo Missions: Have a look! 16
1.3 What's up with those "(Lua Only)" remarks? 16
1.4 DISCLAIMER 16
1.5 Copyright 17
1.6 Acknowledgements 17
2 Introduction 18
2.1 Functional Overview 20
2.1.1 Zone-based Enhancements for ME 20
2.1.2 Feature Enhancements 25
2.1.3 Foundation Level (Lua Only) 27
2.2 Architecture (Lua Only) 30
3 Using DML 32
3.1 How to import Modules into a Mission 32
3.2 Important Concepts 34
3.2.1 Zones and Attributes 34
3.2.2 Flags in DML 35
3.2.3 Understanding Flag Use 38
3.2.4 Multiple Output Flags 40
3.2.5 Using Flags as "signal lines" 40
3.2.6 Attribute Synonyms – Why? 41
3.2.7 DML Flags: Named and Local Flags 42
3.2.8 DML Flags and access via API / global access to local Flags 44
3.2.9 Module Configuration Zones 44
3.2.10 Orders 45
3.2.11 Spawn Formations (spawners) 46
3.2.12 Spawning: Type String and Type String Arrays 47
3.2.13 Ownership / Owned Zones 48
3.3 DML Mission Design Philosophy (Lua Only) 49
3.3.1 Skel: The most basic mission script 49
3.3.2 Game States 50
3.3.3 Update() Loop 50
3.3.4 Event Handler 50
3.3.5 ME-provided "Configuration Zones" 51

3.3.6 The Start() Method ... 52
3.3.7 The “Main” Skeleton ... 53
3.4 Using Zone Enhancements ... 55
3.4.1 All Zone Enhancements ... 57
3.4.2 cfxSmokeZones ... 58
3.4.3 rndFlags ... 61
3.4.4 pulseFlags ... 68
3.4.5 delayFlags ... 71
3.4.6 Raise Flag ... 73
3.4.7 xFlags (Flag Combining / Testing) ... 75
3.4.8 countDown ... 82
3.4.9 cfxObjectDestructDetector ... 86
3.4.10 cfxSpawnZones ... 89
3.4.11 Clone Zones ... 94
3.4.12 cfxObjectSpawnZone ... 105
3.4.13 cfxCargoReceiverZone ... 110
3.4.14 cfxArtilleryZones ... 113
3.4.15 cfxOwnedZones ... 117
3.4.16 FARP Zones ... 122
3.4.17 cfxMapMarkers ... 126
3.4.18 cfxNDB ... 127
3.4.19 Messenger ... 131
3.4.20 Unit Zone ... 134
3.4.21 Group Tracker ... 139
3.4.22 Wiper (Unit/Object removal) ... 144
3.4.23 Radio Trigger ... 148
3.4.24 baseCaptured ... 150
3.5 Using Stand-Alone Features ... 152
3.5.1 Player Score ... 154
3.5.2 cfxHeloTroops ... 158
3.5.3 jtacGrpGUI (tbc) ... 161
3.5.4 csarManager ... 162
3.5.5 Limited Airframes (tbc) ... 167
3.5.6 Guardian Angel ... 168
3.5.7 Parashoo ... 172
3.5.8 Civ Air ... 173
3.5.9 Artillery UI ... 178

3.5.10 Recon Mode 181
3.5.11 ssbClient 186
3.5.12 ssbSingleUse (tbc) 190
3.5.13 unGrief (tbc) 191
3.5.14 cfxMon Development Tool (Lua Only) 192
3.5.15 Module Name 194
3.6 Using Foundation (Lua Only) 195
3.6.1 dcsCommon (Lua only) 195
3.6.2 cfxPlayer (Lua Only) 199
3.6.3 cfxZones (Lua Only) 205
3.6.4 cfxCommander (Lua Only) 209
3.6.5 nameStats (Lua Only) 215
3.6.6 cargoSuper (Lua Only) 219
3.6.7 cargoManager (Lua Only) 224
3.6.8 cfxGroundTroops (Lua Only) 227
3.6.9 cfxGroups tbc (Lua Only) 233
3.6.10 cfxMX tbc (Lua Only) 233
4 Foundation API 235
4.1 dcsCommon API 236
4.1.1 Miscellaneous Methods 236
4.1.2 Table / String Managements 239
4.1.3 Vector Math 241
4.1.4 Airfield, Landable Ships and FARP 242
4.1.5 Group handling 244
4.1.6 Unit Handling 245
4.1.7 Spawning Units / Group, Routes, Tasks 247
4.1.8 Static Objects 250
4.1.9 Coalition 250
4.1.10 Event Handling 251
4.2 cfxPlayer API 252
4.2.1 Tables 252
4.2.2 Callback Handling 252
4.3 cfxZones API 254
4.3.1 Testing 255
4.3.2 Management 257
4.3.3 Properties 259
4.3.4 Spawning 262

4.3.5 Miscellaneous 262
5 Tutorial / Demo missions 264
5.1 Overview 264
5.2 Smoke'em! DML Intro.miz 268
5.2.1 Demonstration Goals 268
5.2.2 What To Explore 268
5.2.3 Discussion 269
5.3 Object Destruct Detection (ME Integration).miz 270
5.3.1 Demonstration Goals 270
5.3.2 What To Explore 270
5.3.3 Discussion 271
5.4 ADF and NDB Fun.miz 272
5.4.1 Demonstration Goals 272
5.4.2 What To Explore 272
5.4.3 Discussion 274
5.5 Artillery Zones Triggered.miz 275
5.5.1 Demonstration Goals 275
5.5.2 What To Explore 275
5.5.3 Discussion 277
5.6 ME Triggered Spawns.miz 278
5.6.1 Demonstration Goals 278
5.6.2 What To Explore 278
5.6.3 Discussion 279
5.7 Spawn Zones (training and lasing).miz 280
5.7.1 Demonstration Goals 280
5.7.2 What To Explore 280
5.7.3 Discussion 281
5.8 Random Glory / Random Death (rndFlag) 283
5.8.1 Demonstration Goals 283
5.8.2 What To Explore 283
5.8.3 Discussion 286
5.9 Pulsing Fun.miz (pulseFlags) 287
5.9.1 Demonstration Goals 287
5.9.2 What To Explore 287
5.9.3 Discussion 288
5.10 Attack of the CloneZ.miz (Clone Zones and Delay) 289
5.10.1 Demonstration Goals 289

5.10.2 What To Explore 289
5.10.3 Discussion 295
5.11 Flag Fun – LOCAL FLAGS (& raiseFlags) 297
5.11.1 Demonstration Goals 297
5.11.2 What To Explore 297
5.11.3 Discussion 299
5.12 Once, twice, three times a maybe.miz (Event Count Down) 300
5.12.1 Demonstration Goals: The ~~un~~ending Spawner 300
5.12.2 What To Explore 300
5.12.3 Discussion 302
5.13 Bottled Messages.miz (Messenger, timeDelay) 303
5.13.1 Demonstration Goals 303
5.13.2 What To Explore 303
5.13.3 Discussion 304
5.14 Follow Me! (unit zones & messenger) 306
5.14.1 Demonstration Goals 306
5.14.2 What To Explore 306
5.14.3 Discussion 308
5.15 Clone Relations (Advanced Topic) 310
5.15.1 Demonstration Goals 310
5.15.2 What To Explore 311
5.15.3 Discussion 314
5.16 Moving Spawners.miz (SpawnZone and linkedUnit) 315
5.16.1 Demonstration Goals 315
5.16.2 What To Explore 315
5.16.3 Discussion 316
5.17 Helo Trooper.miz 317
5.17.1 Demonstration Goals 317
5.17.2 What To Explore 317
5.17.3 Discussion 319
5.18 Helo Cargo.miz – cargo spawn & receive 321
5.18.1 Demonstration Goals 321
5.18.2 What To Explore 321
5.18.3 Discussion 322
5.19 Artillery with UI.miz 324
5.19.1 Demonstration Goals 324
5.19.2 What To Explore 324

5.19.3 Discussion 326
5.20 Missile Evasion (Guardian Angel).miz 328
5.20.1 Demonstration Goals 328
5.20.2 What To Explore 328
5.20.3 Discussion 328
5.21 Recon Mode.miz 330
5.21.1 Demonstration Goals 330
5.21.2 What To Explore 330
5.21.3 Discussion 331
5.22 Owned Zones ME Integration.miz 332
5.22.1 Demonstration Goals 332
5.22.2 What To Explore 332
5.22.3 Discussion 333
5.23 FARP and away.miz (tbc) 334
5.23.1 Demonstration Goals 334
5.23.2 What To Explore 334
5.23.3 Discussion 334
5.24 Keeping Score.miz (PlayerScore) 335
5.24.1 Demonstration Goals 335
5.24.2 What To Explore 335
5.24.3 Discussion 336
5.25 The Zonal Countdown (Local/Global Flag Demo) 337
5.25.1 Demonstration Goals 337
5.25.2 What To Explore 337
5.25.3 Discussion 339
5.26 Frog Men Training.miz 341
5.26.1 Demonstration Goals 341
5.26.2 What To Explore 341
5.26.3 Discussion 344
5.27 CSAR of Georgia.miz 345
5.27.1 Demonstration Goals 345
5.27.2 What To Explore 345
5.27.3 Discussion 348
5.28 Track This! (Group Tracker) 349
5.28.1 Demonstration Goals 349
5.28.2 What To Explore 349
5.28.3 Discussion 351

5.29 Watchflags Demo 352
5.29.1 Demonstration Goals 352
5.29.2 What To Explore 353
5.29.3 Discussion 356
5.30 Viper with a double youu (Wiper) 357
5.30.1 Demonstration Goals 357
5.30.2 What To Explore 357
5.30.3 Discussion 359
5.31 Radio Go Go (Radio Trigger) 361
5.31.1 Demonstration Goals 361
5.31.2 What To Explore 361
5.31.3 Discussion 362
5.32 xFlags – Field Day (Decisions, Flag Testing) 363
5.32.1 Demonstration Goals 363
5.32.2 What To Explore 363
5.32.3 Discussion 365
5.33 Virgin (Civ) Air / Air Caucasus II / One-Way Air (CivAir) 367
5.33.1 Demonstration Goals 367
5.33.2 What To Explore 367
5.33.3 Discussion 369
5.34 Count Bases Blue (baseCaptured, xFlags) 371
5.34.1 Demonstration Goals 371
5.34.2 What To Explore 371
5.34.3 Discussion 372
5.35 DML Mission Template.miz – (Lua Only) 374
5.35.1 Demonstration Goals 374
5.35.2 What to explore 374
5.35.3 Discussion 375
5.36 Landing Counter.miz – (Lua Only) 380
5.36.1 Demonstration Goals 380
5.36.2 What To Explore 380
5.36.3 Discussion 380
5.37 Event Monitor.miz (tbc) 382
5.37.1 Demonstration Goals 382
5.37.2 What To Explore 382
5.37.3 Discussion 382
5.38 Mission.miz 383

5.38.1 Demonstration Goals 383
5.38.2 What To Explore 383
5.38.3 Discussion 383
6 Mission Design FAQ 384
6.1 General 384
6.1.1 Loading Modules 384
6.1.2 Module Misbehaving? 384
6.2 Module Specific 384
6.2.1 Messenger 384
6.2.2 Object Destruct Detector 385
6.2.3 SSBClient 385

cf/x

Dynamic Mission Library
for DCS

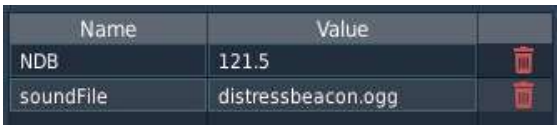
PART I:

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.

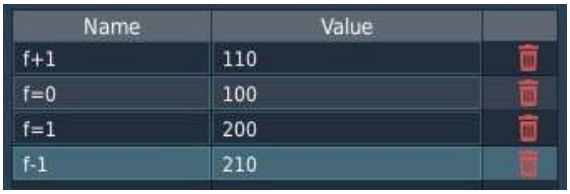
So, what is DML? It's a **mission-building toolbox that does not require Lua,** yet also provides comprehensive support if you do want to use Lua. At its heart are modules that **attach themselves to Mission Editor's (ME) Trigger Zones** to provide new abilities. Mission designers control abilities in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



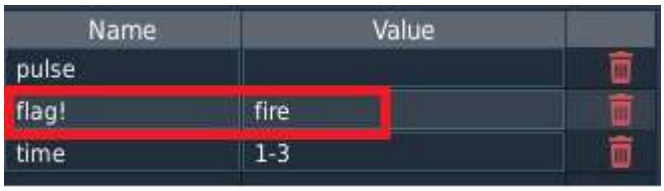
For example, when you add above attributes to a trigger zone, the "cfxNDB" module automatically activates for this zone, and starts an NDB at the zone's center at 121.5 MHz, playing the "distressbeacon.ogg" sound file on that frequency.

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.

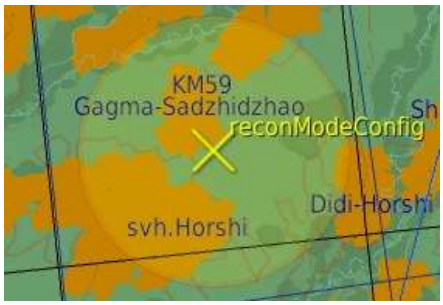


If that isn't enough, DML modules **can be triggered with ME flags**, while **others can set ME Flags** when they activate. For example, spawn zones can be instructed to watch flag 100, and spawn every time when that flag changes its value. Other modules can be told to increase a Flag (e.g., 110) every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



As **a significant expansion** to classic mission design, **DML supports named flags,** i.e. flags that have name like 'bunker destroyed' rather than a number (like 1234). Since early 2022, DCS ME *also* supports named flags, so it has achieved parity with DML in that respect. DML goes a bit further with zone-local flags, Watchflags and multiple simultaneous flag changes, and you are encouraged to explore. DML is 100% compatible with ME flags.

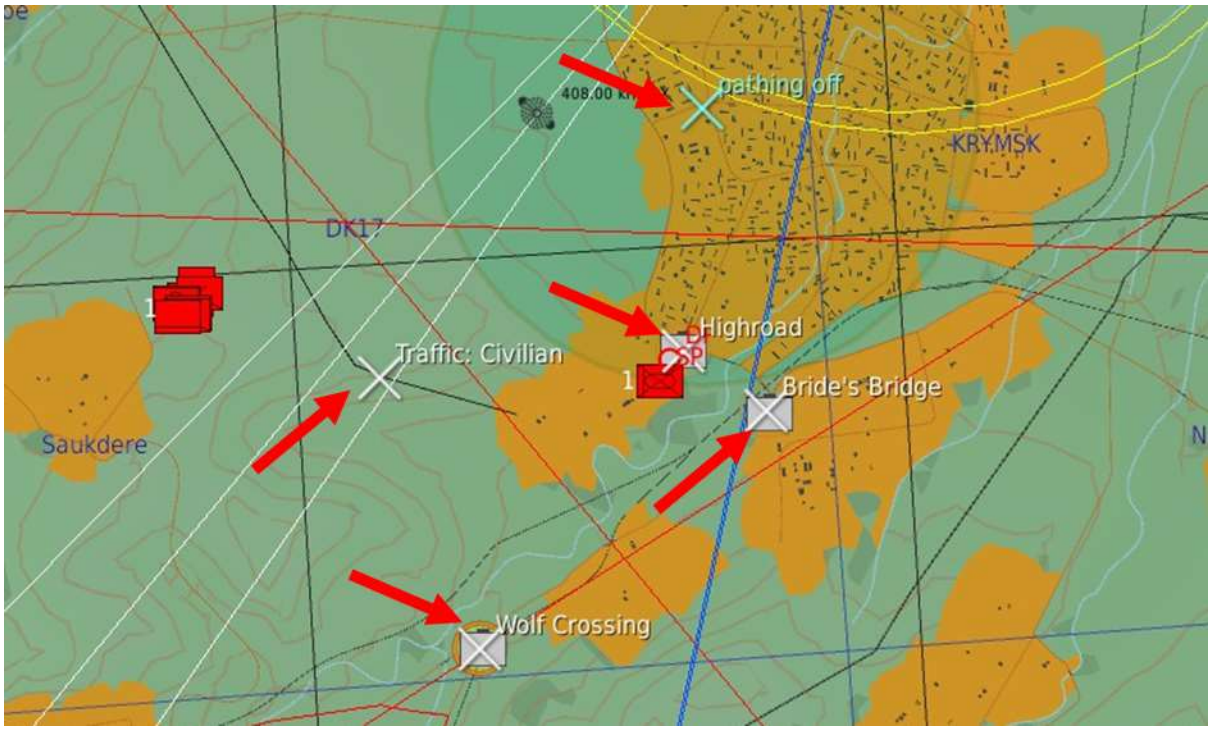
If a module requires configuration data, it starts up with default values, and then looks for a – surprise! – Trigger Zone that might contain the attributes that you want to change for this mission. **You can configure your modules from within ME** – you don't have to change a single line of code.



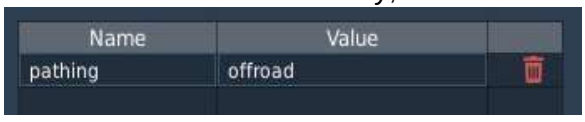
DML has something in store for every mission designer – novices and veterans alike. And for mission designers who have discovered Lua, DML can super-charge their abilities. That being said, **Lua knowledge is not required** to use DML in your missions. At all.

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.



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
    - Griefer Protection
    - 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
    - Comprehensive Flag manipulation and handling like Flag Randomizers and Flag Pulsers so you can completely randomize troop activation and spawning
    - Counters for events
    - Group Tracking and Unit Detection
    - Dynamic Ground Troop Spawning and Cloning
    - Dynamic Object/Cargo Spawning
    - Artillery Target Zones
    - Conquerable Zones and FARPS
    - (moving) NDB
    - Map/Scenery object destruction detector

- Support for **named and zone-local flags** so you can finally use flags with names like ‘Tanker on Station’ instead of ME’s old-school ‘1337’ number-only names (which are only cool once in a while).

- **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 / Demo Missions** 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 and 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 comprehensive, fun 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

I would like to acknowledge the following people and their efforts:

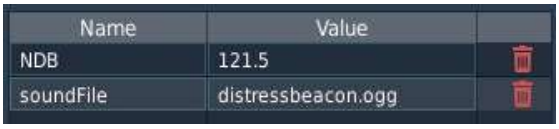
- cloose for their work on the baseCaptured module (initial idea and code proposal)

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**
  Enhancements use Mission Editor's Trigger Zones to attach ("anchor") themselves, and "Attributes" to control how these new features work. 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). In the example to the right, the "NDB"-named attribute tells DML that this zone anchors the *cfxNDB* module. The value "121.5" for NDB tells cfxNDB that this beacon should transmit on 121.5 MHZ, while the "soundFile : distressbeacon.ogg" attribute details which sound file is to be played over that frequency. You can find out which attributes are required or optional for which module in the "Using DML" part of this documentation.

  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 rndFlags

You new best friend. A simple way (finally!) to **randomly set ME flags**, with tight control over which flags should be set, when, and how. After using these you won’t believe that you ever had to put up with the old way.

2.1.1.4 pulseFlags

A module that **repeatedly sets ME flags** – multiple times, for as many times as you want, at your own pace and time, with (of course) controllable start and stop commands (via flags)

2.1.1.5 delayFlags

Smaller sibling to pulseFlags this module simply introduces a **delay before it changes** a flag.

2.1.1.6 RaiseFlags

A module to **set flags to a value** at mission start or a (randomizable) time after start.

2.1.1.7 xFlags

A module to **combine multiple flags for decision making** (‘one of these flags is true?) or to use as a **‘gated switch’** (i.e. a switch that itself can be turned off)

2.1.1.8 CountDown

A very flexible **counter module that counts changes on flags**.

2.1.1.9 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.

2.1.1.10 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.

2.1.1.11 Clone Zones

Like Spawn Zones this module adds the ability to dynamically create units. However, Clone Zones create groups from templates that are easy to build in ME. Spawning occurs on demand.

2.1.1.12 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 useOffset to make objects spawn relative to the ship’s center point)

2.1.1.13 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

2.1.1.14 cfxArtilleryZones

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

2.1.1.15 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).

2.1.1.16 FARPZones

Adds the ability to **make FARPs conquerable** like Owned Zones, with easy placements of defendes, 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.

2.1.1.17 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

2.1.1.18 cfxNDB

Adds an NDB to Zones. The NDB **can move with a unit**, enabling mission designers to easily place NDB on ships that move with it. NDBs can be **turned on and off** at will **using ME flags** or API.

2.1.1.19 Messenger

A DML version of good old MESSAGE TO ALL and SOUND TO ALL – with flags to trigger and some (limited) variable reading: **send text and audio messages from zones**.

2.1.1.20 Unit Zone

Trigger Zones done right: set all the attributes right there on the zone and drive flags the way god wanted you to.

2.1.1.21 Group Tracker

Have DML **watch your groups** and set flags when something changes.

2.1.1.22 Wiper

A DML module to remove objects, units and other stuff inside the zone.

2.1.1.23 Radio Trigger

Some nice interface glue to use “Communication” radio bound flags with DML modules

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.

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 so 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

DML **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 modules 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 “primitves” 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 parlor), 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'**.

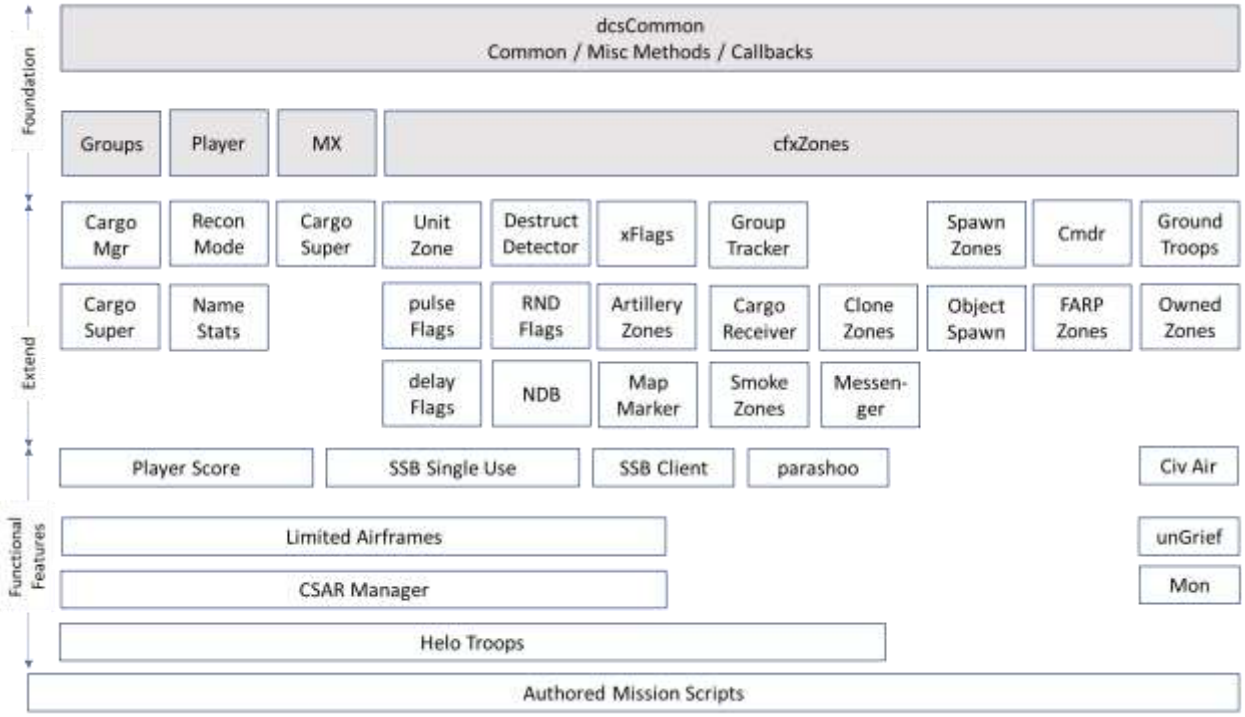
2.1.3.10 cfxMX

A module that reads the mission data at start to provide **information about all groups** that are defined at the time that the mission starts (i.e. created with ME). Cannot provide information about groups that are spawned after the mission starts..

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.

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:

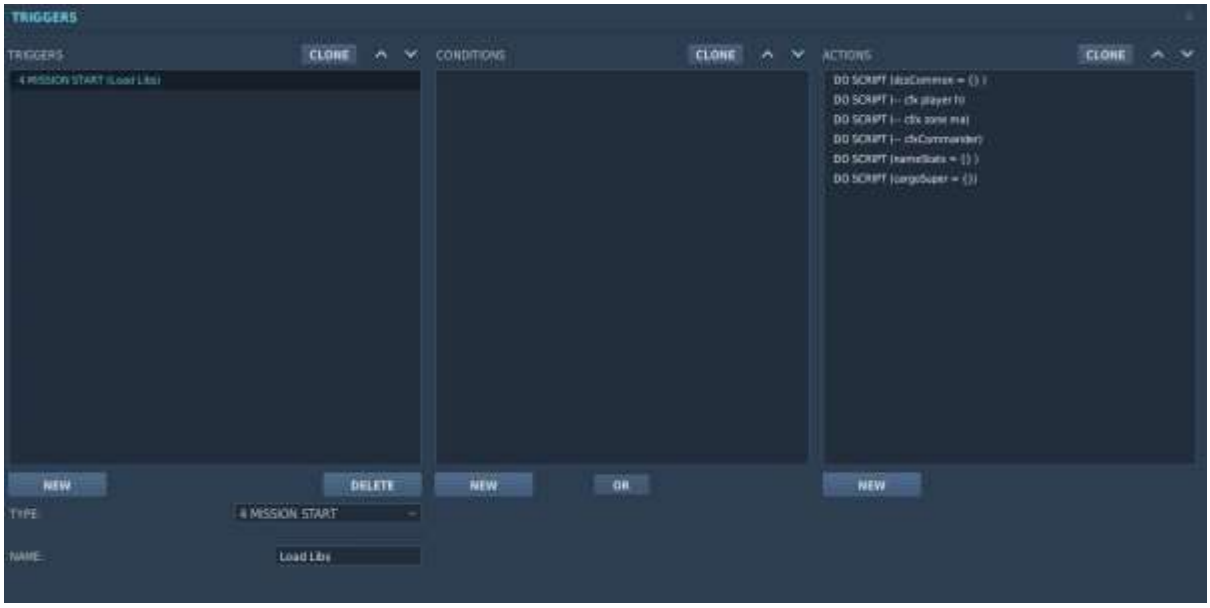
- *Importing Modules into Mission*
  Describes the concept of what modules are, and how you import modules into your mission.

- *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

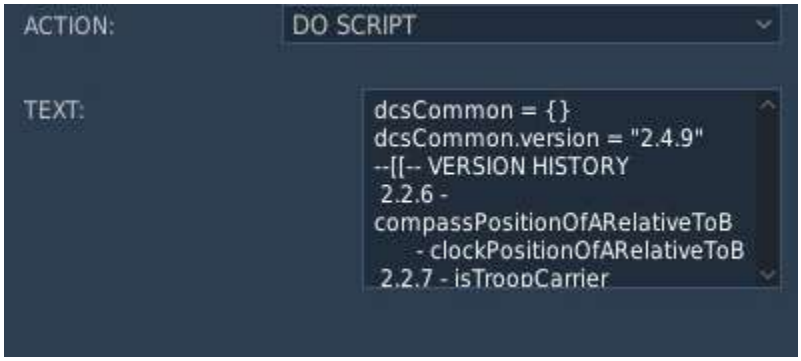
DML is organized in multiple “modules” which are nothing more than small text files. Each of these text files is named after the module they contain. To bring these modules into your mission, first create a MISSION START trigger. Then add DO SCRIPT actions to that trigger, one for each module that you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

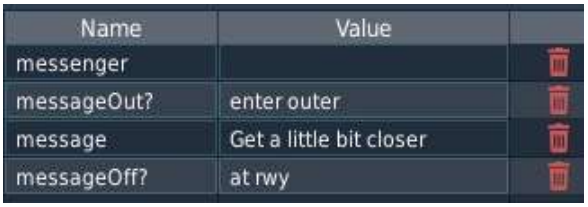
Usually, the name you give to a Zone (“Red Two”) itself is irrelevant; DML looks for specifically named attributes to anchor a module. You can therefore use the same zone to anchor multiple modules.

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.

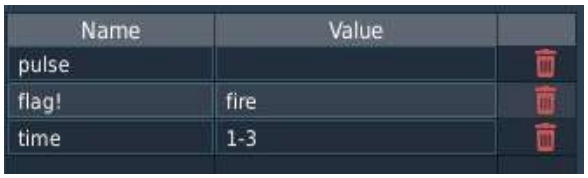
3.2.2 Flags in DML

Beside Trigger Zones, another central ME tool allows mission designers to control the flow of control: 'Flags'. Mission designers use Flags to "persist" (remember) states. Although Flags in ME are somewhat primitive, they can be used to great effect – as many existing missions show.

Many Zone Enhancements in DML can monitor ("watch") a flag ("Input") for a value change which can then trigger the module's action (in DML parlance, we call these "Watchflags"). Others can change a flag's value ("output") when they determine that certain conditions are met (for example a group that they watch has died, or a zone is conquered).



You tell a module which flags it should use for input or output via Zone Attributes. For example, the 'Messenger' module can be triggered via input to display a message, and another trigger can tell it turn off completely. Since triggering is done by flags, you decide which flags to use to use for these purposes, and then list them as values for Messenger's 'messageOut?' and 'messageOff?' attributes. That zone's messenger then triggers the appropriate action whenever these flags' values change.

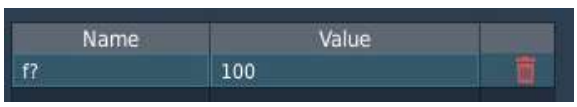


Conversely, many Zone Enhancements can determine that they want to signal something. For example, pulser sends out a repeating signal for other zones to synchronize their actions, or a conquerable zone may want to announce that its ownership has changed; an object destruct detector has detected destruction, or a cargo delivery zone needs to alert the mission that a helicopter has delivered some cargo. Each time, the module can announce this by changing the value of an output flag. Like with the input flags, output flags, you tell the module which flag to change with a Zone Attribute in ME (in above example, the pulser's output flag is an ME flag named "fire").

3.2.2.1 DML "Watchflags"

Many modules offer functionality (e.g. spawn a group, start smoke, output a message) that you want to trigger at exact moments in your mission. To signal a module to activate its functionality, mission designers use ME flags. DML allows mission designers great flexibility when it comes to decide how triggers a function. For example, a mission designer can tell a cloner only to spawn new clones when the value of the flag named "killCount" is greater than 5.

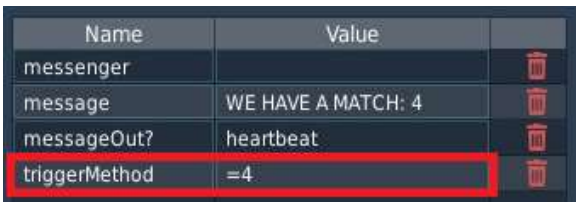
Watchflag Attribute (Input)



Whenever you see a DML module that supports an attribute name that ends on a question mark (e.g., "f?") that means that this module can watch an ME Flag. You supply the flag's name as value (here "100", an old-style ME flag name. Newer flags can have names like "EnemySighted"). If the value of that flag changes, the module's function (e.g., spawning) can trigger, provided all your conditions are met. DML supports a host of conditions that you can apply to flags to trigger selectively (e.g. when the flag's value equals a certain value), or simply trigger on a change (default behavior).

Trigger Methods
After you tell a module which flag to watch, to actually make it trigger at least one of two things must happen:

1. The flag's value must change. DML Watchflags only trigger on flag value changes and ignore the flag otherwise
2. (Optional) The new value of the watched flag must match the rule (condition) that you describe in the module's trigger method attribute



Why is the second point optional? DML assumes that simply changing the Watchflag's value is enough to trigger the module's function. So, when you give a zone's module a Watchflag and skip telling DML what exactly to look for, it will simply activate whenever that flag's value changes. That simple mechanism will cover most of your mission demands: all enemy tanks are destroyed; this triggers a change in a flag, and that flag is wired into your cloner → a new clone group spawns!

Sometimes, however, you want finer control over when something happens, and this is where the trigger "methods come" in. These are simple, predetermined rules that are applied to the current value of the Watchflag. For example, the rule "=4" means that in order to trigger, the Watchflag must not only change, but after the change, the flag's value must also be equal to the number 4.

DML supports multiple comparisons, and even allows you to compare the Watchflag to other flags that you can name in the trigger method. DML understands the following rules:

- `'change'` or `'#'`
  triggers whenever the Watchflag's value changes

- `'off'` or `'0'` or `'no'` or `'false'`
  triggers when the Watchflag's value changes to zero

- `'on'` or `'1'` or `'yes'` or `'true'`
  triggers when the flag's value changes from zero to non-zero
  (**Warning**: DML will not detect a transition between two non-zero numbers e.g., 3→4, it only triggers on a change from ZERO to a non-zero value)

- `'inc'` or `'+1'`
  triggers when the Watchflags's value changes to exactly the Watchflag's previous value plus one

- `'dec'` or `'-1'`
  triggers when the Watchflag's value to exactly the Watchflag's previous value minus one

- `'>(number)'` or `'>(name)'`
  triggers when the Watchflag's value changes, and the value is larger than the number given or flag identified by name

  Examples:

- `>4` triggers when the Watchflag's value is larger than the number 4
- `>*landings` triggers when the Watchflag's value is larger than the value of local flag 'landings'

- `'=(number)'` or `'=(name)'`
  triggers when the Watchflag's value changes, and the value is equal to the number given or flag identified by name

  Examples:
  - `=4` triggers when the Watchflag's value is equal to the number 4
  - `=*landings` triggers when the Watchflag's value is equal to the value of local flag 'landings'

- `'<(number)'` or `'<(name)'`
  triggers when the Watchflag's value changes, and the value is less than the number given or flag identified by name

  Examples:
  - `<4` triggers when the Watchflag's value is less than the number 4
  - `<*landings` triggers when the Watchflag's value is less than the value of local flag 'landings'

- `'#(number)'` or `'#(name)'`
  triggers when the Watchflag's value changes, and the value is not equal to the number given or flag identified by name

  Examples:
  - `#4` triggers when the Watchflag's value is not equal to the number 4
  - `#*landings` triggers when the Watchflag's value is not equal to the value of local flag 'landings'

Quoting Numbered Flags
Early versions of DCS used only numbers for flag names. For example, "22" was (and still is) a legal flag name. This can create confusion when using Watchflags as it can't tell the difference between a number and a flag whose name happens to be a number.

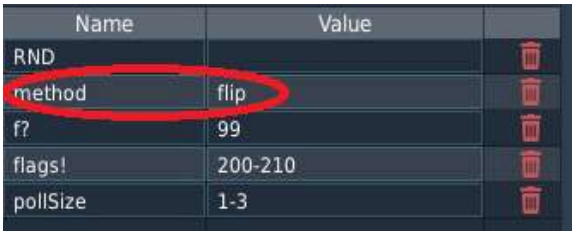
In order to allow DML to distinguish between a number and **flag whose name happens to be a number**, such a flag's name **must be put into double quotes** '"' and '"' to be interpreted as a flag number. Hence, if you want to trigger only if the input flag was equal to flag named 22, the condition would be

="22"

DML then (and only then) recognizes "22" as meaning the flag named 22 rather than the number 22.

3.2.2.2 Bang! Method (output)

Most modules support a variable, mission designer selectable bang! ‘method’.



DML understands a number of bang! methods that it applies to the flag. Currently, DML understands the following methods, each defined by a keyword:

- ‘on’
  Sets the flag’s value to one, no matter what it was before. Same as using the number 1 (one)

- ‘off’
  Sets the flag’s value to 0 (zero), no matter what the value was before. Same as using the number 0 (zero)

- ‘inc’
  Increases the flag’s value by 1 (one). If, for example, the flag’s value was previously 10, that is increased to 11

- ‘dec’
  Decreases the flag’s value by 1 (one). If, for example, the flag’s value was previously 10, this number is decreased to 9.

- ‘flip’
  This is the most effecting method if you intend to trigger on flag change. It flips the flag’s value between 0 (zero) and “not 0”: If the flag’s value was anything except zero, the new value is zero. If the flag’s value was zero, the new value is 1 (one). This way you can flip-flop flags, turning them on and off repeatedly.

- (number)
  Sets the flag to the fixed value (number), no parentheses. Example: 33 – sets the flag’s value to (the number) 33

3.2.3 Understanding Flag Use

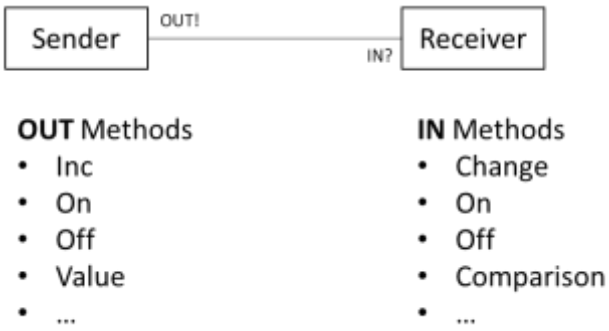
As we have seen above, DML provides two separate ways to communicate with other modules:

- The way it send signals (the **output method**, usually designated by the exclamation point "!" at the end of the attribute's name). It describes what is sent when the module wants to convey information. This can be 'inc' (the most common, increases the flag's value), or 'On', 'Off', 'flip', a fixed value etc. A module sets an output flag whenever it sees fit, usually as a response to something happening in the mission (a group enters a zone, an object is destroyed etc)

- The way a module looks at their inputs (flag attributes with question mark "?" in their name) and tries to decide if it should trigger. Most common is the 'change' method: the module triggers when the flag's current value changes.

Note:
Modules look at their input flags regularly, usually once a second (usually set globally for the module with the ups attribute in the config zone). This means that they do not immediately detect a signal/change, *only the next time they look at their input flags*. This means that they can even miss a signal if the input flag changes too quickly.

Let's put this together: say you have two modules: *'Sender'* that sends signals on 'out!', and *'Receiver'* that looks for a signal on 'in?'. Now, let's look at some combinations to see how output and input methods work together.



- The most common case is *Sender's* output (“DML Method”) is set to 'inc' (increment: add the number 1 to the flag’s current value) and *receiver’s* input (“Watchflag”) to 'change'. Any change of the flag's value made by *Sender* then results in *Receiver* to detect the change next time it checks, and then to trigger.

- Now let’s change the receiver's Watchflag method to '>4', and assume that at the start of the mission, the underlying flag's value is zero (0). *Sender's* method stays at 'inc' (increment by one). The first four times that *Sender* sends a signal, the flag's value changes from 0 to 1, 2, 3 and then 4. Each time that the flag’s value changes, *Receiver* sees the change when it checks, but ignores it. The fifth's time, however, *Sender* has set the flag's value to 5. Upon inspection, *Receiver* detects the change to the flag’s value, sees that the new value is ">4" and triggers. Since modules check input flags regularly, it will **not trigger the next time it checks the flag (usually a second later) because** although the flag's value is >4**, it did not change** from the previous check. **Watchflags require a change** on the watched flag in order to trigger. When *Sender* again changes the flag's value, this time to 6, *Receiver* again triggers. And will trigger every time **when *Receiver* sees that the flag's value has changed from the last time it checked**.

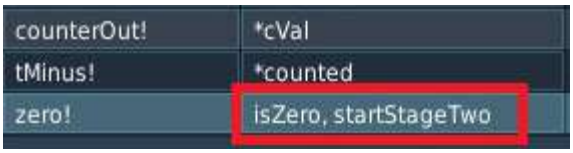
- And now let's set *Receiver's* Watchflag back to "change", while *Sender* is set to 'On'. Now, the first time *Sender* triggers, it sets the flag to On (1). *Receiver* detects the change, sees the new value '1', and triggers. Next time *Sender* is triggered, however, it again sets the flag to On (1). The problem is, though**: it already was set to On (1) before**. *Sender* does not mind; it has done its job. *Receiver* sees that the flag has a value (1 = On), and since there was no change to the previous value (which was also 1), it ignores the current value and continues waiting for a change.

So here's the big take away: **Watchflags are inspected regularly and trigger on change**. If there is no change, the Watchflag does not trigger.

Consider: If multiple modules use 'On' as their output and they all connect to the same flag, only the first flag to set 'On' will subsequently trigger a listening module that is waiting for 'change'. After that, the flag stays on the 'On' value (1). When other modules trigger, they also set that flag to 'On'. To the listening receiver, though, the value of the flag stays the same, and no change is detected; nothing is triggered. This is intentional and immensely helpful to set up common scenarios: Like a ‘Burglar Alarm’ you are able to trigger a central reaction (Alarm) exactly once from multiple possible triggers: the first triggers a reaction, all others no longer elicit a response even if triggered.

To make this perhaps a bit clearer, and to show just how potentially broad this concept of triggering on flag is: A receiver is connected to the IN? flag. That is to say, it is **NOT** connected to a particular sender module. Receiver doesn't care nor know about senders. All it concerns itself with is the value of the IN? flag, and when it changes. So you can (and often do) have multiple modules connect their OUT! ports to same flag. And you can have (and often do) multiple modules connect their IN? ports to the same flag. All senders and receivers work independently from each other, some can be connected or disconnected at will, and they will work together without ever knowing about each other. Only one thing is certain: if any sending module changes the out! flag, all listening receivers will react, and otherwise ignore the current value.

3.2.4 Multiple Output Flags



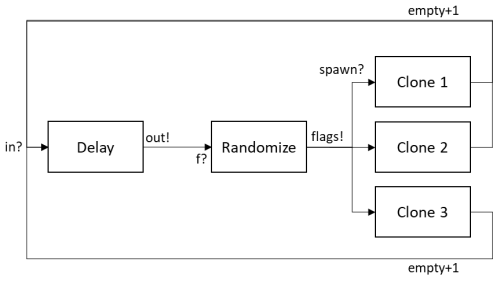
DML has the ability to bang! multiple flags at the same time. Unless otherwise specified, all output bang! flags (those that end on an exclamation point “!”) support this ability. To bang! multiple flags, they need to be listed as the attribute’s value and are separated by a comma; leading/trailing blanks are ignored.

Please note the following:

- All flags are banged with the same method
- All flags are set simultaneously, meaning that there is no guaranteed order in which they are changed.

3.2.5 Using Flags as “signal lines”

In DML, you can greatly simplify your designs if you interpret flags as lines (wire) that transmit signals, reminiscent of old-school digital circuit design. Look at modules like a black boxes with inputs (attributes with ‘?’) and outputs (attributes with ‘!’). Use flags as the wires that connect input and output.



You can then easily design your mission functions as small logical circuits that show the flow of control and greatly reduce complexity.

3.2.6 Attribute Synonyms – Why?

Many modules offer multiple different names ('Synonyms') for the same input signal. For example, a clone zone can use the name *'spawn?*', '*in?*' and '*f?*' all to trigger a new spawn cycle. When you set up your spawn zone, you can use one, *and only one* of these per Zone to trigger a cloning cycle.

Cool - but *why*?

The reason for this is to facilitate module stacking. In DML 'module syntax', most modules support the generic '*f?*' or '*in?*' as input. This makes sense, as it's easy to remember, and often, if you stack modules on the same zone, you want them to trigger at the same time. Use a single '*f?*', and all anchored modules that understand '*f?*' as input will trigger when that flag changes.

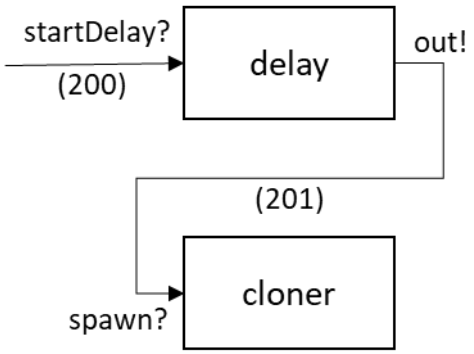
In the example on the right, we have two modules that stack on the zone: a cloner, and a messenger. Both cloner and messenger support '*in?*' as input flag (here set to flag 401). When flag 401 changes, both the cloner and the messenger modules activate and do their thing.

Name	Value
cloner	
in?	401
messenger	
message	Convoy sighted at bridge!

But what if you want to stack two modules that share the same input name, but you *don't* want them to activate at the same time? A common case is if we build a small 'circuit' directly on the zone. The example on the right uses a delay module to delay the incoming signal on flag 200 by 2 seconds before passing it '*out!*' on 201. Since 201 is 'wired' into the cloner's '*spawn?*' input, this causes the cloner to spawn 2 seconds after the timeDelay received a signal.

Name	Value
cloner	
spawn?	201
timeDelay	2
out!	201
startDelay?	200

If we look at delayFlag's and cloner's documentation, we'll find that they both share '*in?*' and *'f?*' as inputs. So we can't use these synonyms as they would cause both to activate. We have to use an input name that we can tell apart: delayFlag supports the uniquely named '*startDelay?*' input synonym (i.e. cloner does *not* recognize this attribute), while the cloner supports '*spawn?*' (which delayFlag does not recognize). If we use the two synonyms that are not shared, we can connect the input lines to different signals, and thus are able assemble this little signal delay 'circuit' on the same zone without getting our attributes crossed.

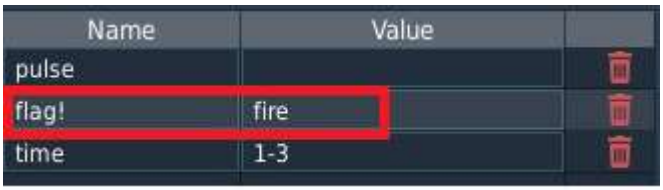


That's why DML supports synonyms.

3.2.7 DML Flags: Named and Local Flags

This is arguably one of DML's most powerful and important features. Put differently, one of ME's less loved aspects was the fact that flags only used numbers as names [NOTE: This has changed with a late version of DCS 2.7 – Now DCS ME also supports named flags]. So it was up to the mission designer to remember that flag "37" is used to signify that the enemy Flanker has been triggered, while "45" might be used to signal that the tanker is on-station.

3.2.7.1 NAMED FLAGS



DML supports named flags. There are some (minor) rules to adhere to (→ Flag Naming Rules), but in DML you can use meaningful flag names! Of course this is backwards-compatible to old-style ME: use a flag name that is entirely a number (e.g. "123"), and this flag is recognized by older versions of DCS ME and can be used with all ME triggers. DML's named flags are fully compatible with the newer, late DCS 2.7+ version named flags.

HISTORICAL NOTE
From 2008 through early 2022, DCS ME supported only flags whose names were positive integer numbers, excluding zero. Since late versions of DCS 2.7, ME is fully compatible with DML-style named flags.

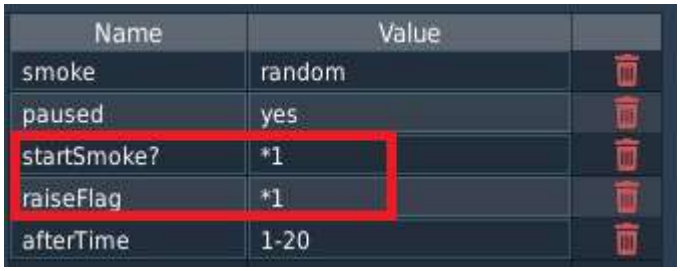
Using flags, you can pass flag signals between DML modules that use good, recognizable names (like 'Tanker On Station') that do not clutter up your ME flag space (which you will probably still use for UI and other mission-related tasks).

Flags are what is used to communicate in ME in general, and between modules DML in particular. Since DML modules make it easy to communicate, flag usage in DML-enhanced mission increases significantly.

*3.2.7.2 ZONE-LOCAL FLAGS: *name*

There is another feature DML feature that, combined with Zones, exponentially raises flag utility: **local flags**. Any mission designer who has spent just a little time designing DML-enhanced missions has assembled some small 'Zone Automations: an assembly of DML modules that talk to each other purely inside the zone: often involving a delay flag, raise flag, messenger, count down, pulse flag or randomizer. The problem with these "automatons" is that they still require one or more unique flags to communicate with each other, and that makes them difficult and ungainly to work with, especially when using copy/paste.

Enter zone-local flags: these flags, recognizable by a name that **begins with an asterisk '*'** is zone-local. This means that a different zone that uses exactly the same name will access it's own zone-local incarnation, and not cross signal with a same-named flag in another zone. In the example on the right, the raiseFlag module bangs flag "*1", which the stacked smoke module's input startSmoke? uses. So, all modules that stack on this zone share the value of *1. To any module attached to a different zone, that flag is not only invisible, that zone may even have it's own *1 flag with a different value. This way, this zone can pass values between its modules without cluttering your mission's flag name space.



3.2.7.3 FLAG NAMING RULES

So how can we distinguish between flags, and how do we use them? Generally speaking, ME-compatible flags must be numbers only, while DML flags can be named anything (with some restrictions: they should not begin with a digit, and must not contain a comma ','). A special subclass of DML flags, the zone-local flags all start with an asterisk, and otherwise have the same restrictions as other DML flags.

	ME	**DML**	**DML Zone-Local**
Format	Positive Number, (new: Alphanumeric)	Alphanumeric, must not contain comma ',', **must not start with asterisk '*', double quote '"' or digit ('0'...'9')**	**Starts with asterisk** '*', alphanumeric, must not contain comma ','
Examples	123 99 2898	A12 With blank F***d up Yup "quotes" too	*1 *A12 *fireCloner *ok multi **
Scope / Visibility	Entire DCS	DML modules, Lua Scripts Entire DCS (newer versions)	Only DML modules in same zone
Invisible to		Old DCS versions	Everyone outside Zone

3.2.7.4 DML FLAG RANGES

There is one case in which old-style ME number-only flags make sense, and I recommend that you make use of it: some modules support flag ranges, e.g. "10-20". This only works with numerical global (old-school ME) flag names, and you must be careful not to accidentally mix these with local flags. Used strategically, tough, numerical flag ranges can be quite comfortable. Mission designers' discretion is advised.

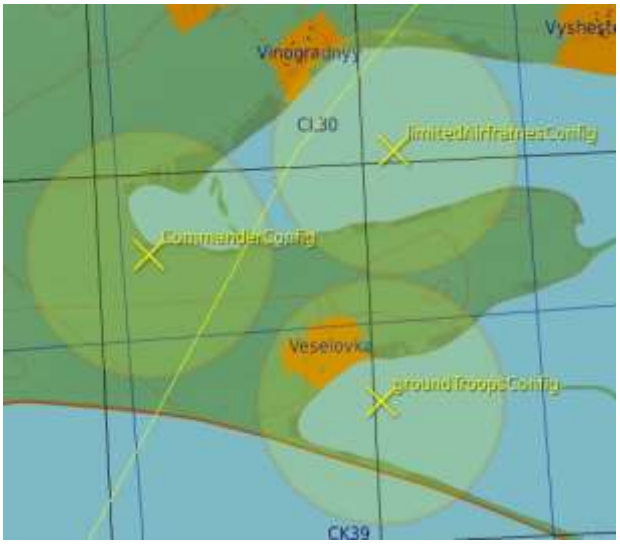
3.2.8 DML Flags and access via API / global access to local Flags

So how are DML flags accessible from Lua? This is luckily quite simple, since DCS Lua scripting always supported named flags in addition to numbered. So if you choose a globally named flag (e.g. “Eagle Landed”) that flag is also available from Lua.

“Local” flags (flags starting with an asterisk “*”, e.g. “*boom*) are also accessible from outside if you observe DML’s local name creation rules. When you designate a flag as local in a zone’s attribute starting the flag name with an asterisk ‘*’, all that DML does is prepend the zone’s name. So, if a zone name “Hog Starting Area” uses a local flag “*hog rolling”, that flag is accessible to Lua (*and from other zones as well!*) by its global name “Hog Starting Area*hog rolling”

3.2.9 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 menitioned, 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.

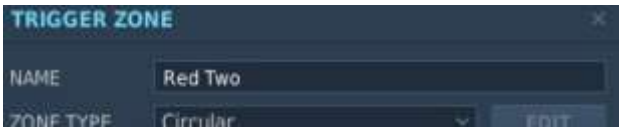
Unlike with most other zones, placement (location) of a configuration zones irrelevant, you can place them anywhere you like. A good place for them is somewhere out of the way where they can’t confuse or get in the way (one of the corners of the map, for example). Some people also like to color-code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

3.2.10 Orders

Ordering troops is an ability multiple modules support in DML: modules that 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.

3.2.10.1 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	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.  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'	target
lase laze	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. Just one of the units lases, the other units are back-ups if the lasing unit is killed. Units that have lasing orders interact with the jtacGUI script by passing target information and alerting players that lasing information is available. Order attribute can be named 'laze' or 'lase'	range

Orders	Description	Parameters
training train dummy dummies	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. Order attribute can be named 'training' or 'train' DO NOT USE AUTOREMOVE with these orders, or you'll have lots of targets - quickly	

3.2.10.2 "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.

3.2.11 Spawn Formations (spawners)

Note:
The following only applies to units being spawned by spawners. If you dynamically spawn units in a mission with CloneZones, those are assembled in the same formation you gave them in ME.

When groups are spawned, they assemble 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. **If there is only one unit, the center of the spawn zone is used as position.** 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)

Formation	Description
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

3.2.12 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.

3.2.13 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)

PLESE 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 Handers* 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 Skel: The most basic mission script

The most basic usable mission script implements two of those ideas: an event handler and an update method. Thus a truly bare-bones mission script looks something like this:

```
skel = {}
function skel:onEvent(event) -- event handler
end

function skel.update()
      -- schedule next update invocation
      timer.scheduleFunction(skel.update, {}, timer.getTime() + 1)
      -- check states and act accordingly
end

world.addEventHandler(skel) -- connect event hander
skel.update() -- start update cycle
```

Above minimal script is a fully functional mission script hat can handle events and updates every second. It doesn't really *do* anything, but above is pretty much what all good mission scripts boil down to. We'll now build and expand on that concept.

3.3.2 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.

3.3.3 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.

3.3.4 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.

3.3.4.1 dcsCommon event dispatcher

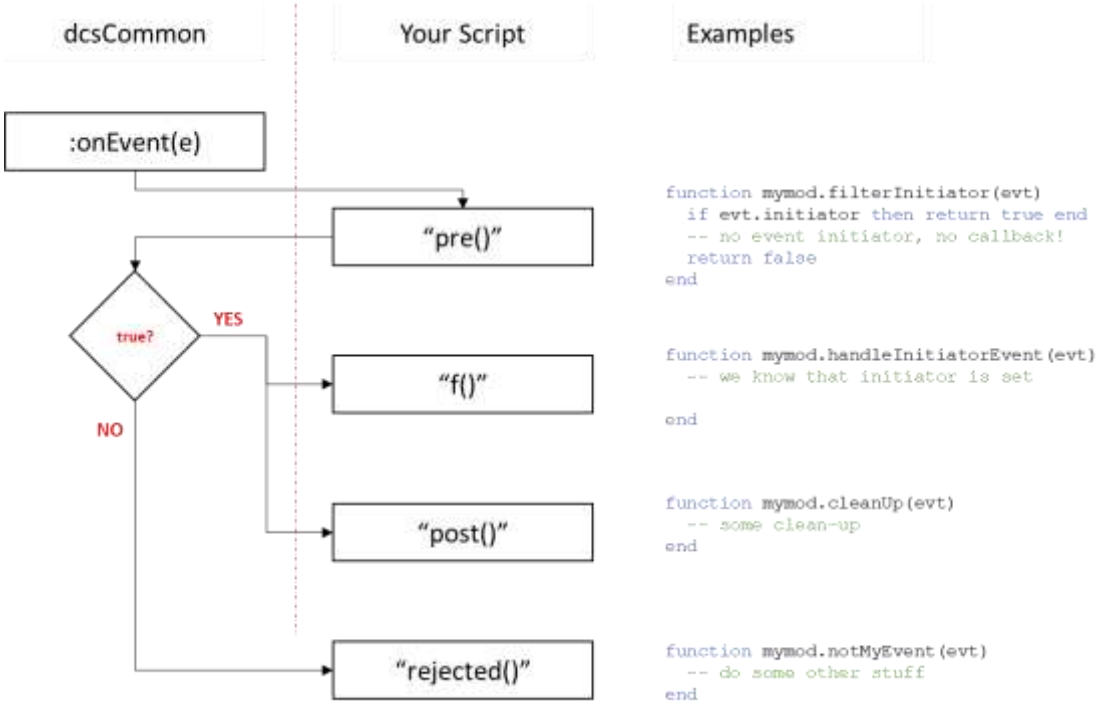
The ‘bedrock’ module dcsCommon (see later) provides a convenient, modular replacement for the one provided by DCS. Instead of implementing your own onEvent() methods on your own tables and calling world.addEventHandler(), you can use dcsCommon’s more advanced event handler that allows for the following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



3.3.5 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)
```

3.3.6 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.

3.3.7 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.

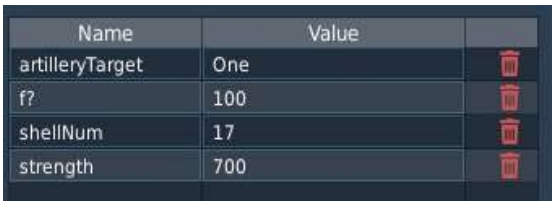
When you look into the code of the various DML modules, you’ll find that they also implement this design: cfxZones uses it to update moving zones, ReconMode monitors aircrafts that way, etc. Most DML modules provide callbacks that you can tie into your own scripted event handlers to be notified when 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: for a practical, more expanded working example of the skeleton code (dmlMain)

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)

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

- **RND Flags**
  A method (attached to a zone) to randomly change a set of flags. Highly flexible, and with great options to make mission randomization much easier

- **Pulse Flags**
  A method (attached to a zone) to repeatedly change a flag in a settable interval.

- **Delay Flag**
  Pulse Flags smaller, more streamlined sibling: when a flag changes, wait for some time, then change another flag.

- **RaiseFlag**
  DML's prime flag raising method. Raise any flag, local or global, at mission start or after a randomizable delay.

- **xFlags**
  Flag manipulation and combine module, often used to create win conditions or decisions.

- **CountDown**
  Counter that counts changes on a flag until its own counter reaches zero. Multitude of versatile options allow you to set this up for many different uses.

- **Object Destruct Detector**
  Changes a ME-compatible flag when a map object is destroyed

- **Ground Unit Spawner**
  Dynamically (in-mission) spawns ground units that can follow DML orders

- **Cloner**
  a.k.a. "Spawner 2.0", this zone can dynamically spawn entire groups, using an ME template (including their template's routes) and works for ground, sea and air units. Unlike Ground Unit Spawners, the cloner does not support DML orders, but uses ME waypoints.

- **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.

- **Messenger**
  A module to display text on the screen, with some niceties added.

- **Unit Zone**
  This module is the DML-version of ME trigger zones, except much more convenient and with support for named and local flags

- **Group Tracker**
  A module to count and track when groups are added to the tracker or are being destroyed.

- **Wiper**
  A module to selectively (or wholesale) remove objects from the game.

- **radioTrigger**
  Provides a convenience DML interface for DCS's communication→radio item triggers that lack a simple reset to be triggered multiple times

- **baseCaptured**
  This module generates signals (changes flags) and provides ownership information for bases (airfields, FARPs, ships)

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 **can** 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

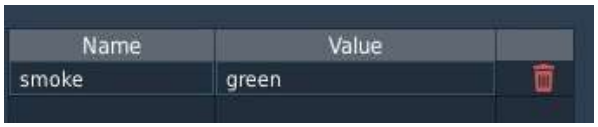
DCS provides a 'smoke' effect in various colors that unfortunately ends after a certain time. For missions it may be useful to be able to *permanently* mark a location with colored smoke.

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 (random, green, blue, white, orange, or red) as value.



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.

ME INTEGRATION

You can start the mission with the smoke disabled and have the smoke effect start when a flag changes value. To do so, first add an attribute "paused" with value "true" (this will cause the smoke to be delayed). Then add an attribute "f?" with the number of the flag (e.g. 100) that you want smokeZones to watch. When this flag's value changes, the smoke starts for this zone.

Name	Value	Description
f? startSmoke?	Name	When the Watchflag triggers, the smoke is started.

Supports DML Flags

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: • "green" or "0" • "red" or "1" • "white" or "2" • "orange" or "3" • "blue" or "4" • "random", "?" or "rnd" (random color from above) **MANDATORY**
paused	When true, will not start smoke at mission start, but wait for a signal on the f? flag. Defaults to false (smoke starts at mission beginning). Note that if you set paused to true and omit the f? attribute, you can only start this smoke via API
f? startSmoke?	Watchflag. When this flag triggers, smoke starts. Defaults to <none>
altitude	Altitude (in meters) above ground that the smoke should be created. Defaults to 1m
triggerMethod smokeTriggerMethod	Conditions when the DML Watchflag should trigger Defaults to 'change'

Supports Watchflags

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, anAltitude, paused, onFlag)

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.

anAltitude is the altitude in meters above ground where the smoke is placed (defaults to 1 meter if omitted), paused will prevent the smoke from starting immediately when true, and onFlag is the number of the ME flag that smokeZones watches for a change to start the smoke.

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.

3.4.2.5.3 startSmoke(aZone)

Starts a paused smoke zone and sets paused to false. If the zone is already playing the smoke effect, another smoke effect is started. aZone can be a string (name of zone) or the cfxZone

3.4.2.6 Using the module

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

Add a zone to the mission, and add the ‘smoke’ attribute to that zone.

3.4.3 rndFlags

3.4.3.1 Description

This module gives you access to what you have been looking for ever since you started writing missions: a simple, flexible and elegant way randomly set flags to make your missions appear more random.

A classic setup is a mission where you have five different groups of enemies, and you want to randomly activate up to any three of them. Doing this with triggers is possible, but nightmarish to maintain and service later.

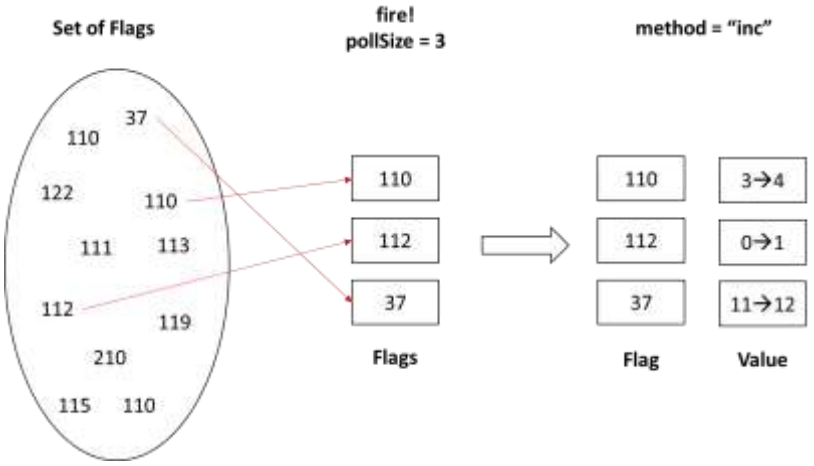
With rndFlags it is just a matter of placing a zone and adding a couple of attributes. rndFlags are exceedingly easy to use, yet incredibly flexible if you need them to be.

Basic Function

rndFlags uses Trigger Zones only to pass data to the randomizer, so you can place the one anywhere on the map.

All rndFlag randomizers function the same way:

- You give it a set of flags (e.g. “100-110, A42”). These are the flags that the randomizer can choose from
- You tell it “fire!”, and the randomizer picks one or more flags from the set and then changes them



Starting from this simple mechanism, rndFlags can also do a lot more if you wish:

- The set of flag the randomizer chooses from is very flexible, and you can add the same flag multiple times
- The randomizer can pick more than one flags at once
- After picking flags, you can remove them from the set so that the next time the set has become smaller. When a randomizer runs out of flags this way, you can tell it to ‘reshuffle’ the deck by putting all flags back
- You have multiple options to tell the randomizer what to do with the flag it chooses
- You can “fire” the randomizer as often as you want

Set of Flags
You start with a set of flags that you pass to the randomizer with the 'flags!' attribute. You list the flags to include separated

- by comma (e.g. "3, G, 4, 5,")
- the order in which flags are listed is not relevant
- you can also include whole ranges by giving the lower and upper bound separated by a hyphen (e.g. "10-20"). This is only applicable for ME-compatible (number-only) flags
- You can mix individual flags and ranges (e.g. "A, 4-8, 9")
- You can add the same flag multiple times (e.g., "9, 4, SomeFlag, 4"). This will add that flag **multiple times** into the set.
- Ranges can overlap with other ranges and individual flags (e.g., "A19, 4-10, 6-12")

Why is it possible to be able to have multiple instances same flag in the set? Because that allows you to increase that flag's likelihood of being chosen over the other flags without adding another layer of calculation to the mix.

Imagine you want flag 5 have a 50% chance to be chosen, and flags 6 and 7 each only a 25% Chance. You achieve that simply by creating a set "5, 5, 6,7" for the randomizer

Poll Size (optional)
Each time the randomizer 'fires' (chooses flags from the set), it takes a number of items from the set of flags. By default, the randomizer chooses a single flag. You can easily change this with the optional pollSize attribute. pollSize understands two different formats as value.

- *Single Number (e.g. 3)*
  The number you give is the 'sample size', the number of items the randomizer chooses from the set of flags. You can set this to any positive number, (if that number exceeds the number of items in the set, the entire set is chosen).
  If for example you choose pollSize = 3, each time you fire the randomizer, three flags are chosen from the set.
  Note that during the pick process, the randomizer makes sure that it does not pick the same *item* twice (of course that does not prevent the same fag to be drawn twice if you have two items that reference the same flag)

- *Range (e.g. 2-5)*
  If you give a range pollSize randomizes the size each time the randomizer fires to a number between (and including) the lower and upper bounds. So, if you set pollSize to "1-3", each time the randomizer fires, pollSize can be 1, 2 or 3.

Remove and Reshuffle (optional)
By default, each time the randomizer fires, the set of flags to choose from is the same. You can change that by enabling the 'remove' option. If set, each time the randomizer fires, the item(s) it chooses from the set are removed. For example, if the start set is "5, 4, 9", and the randomizer picked the "4" item, the next time the randomizer is fired, it uses a reduced set with the remaining items "5, 9".

This continues until the randomizer runs out of flags. When the randomizer runs out of flag, it looks if you also set the 'reshuffle' option. If reshuffle is set, the randomizer "reshuffles" all flags by returning to the original full set of flags that it started with.

If reshuffle is disabled, it simply does nothing, as all flags are exhausted, and there is nothing to do.

Flag Change Method (optional)
After the randomizer has chosen the flag(s), it changes them according to standard DML Flag methods.

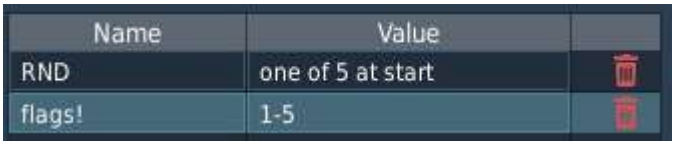
fire! – running a randomizer cycle
There are multiple ways to have a rndFlag randomizer run a cycle:

- `onStart` (optional)
  A randomizer that has the onStart attribute set to true will run the randomizer 0.25 seconds after the mission starts. The 0.25 second delay is intentional to allow all other modules to initialize and the mission itself to 'settle' before DML starts polling and changing flags.

- `f?`
  This is a DML watchflag. The randomizer watches this flag for change, and each time the value of this flag changes, the randomizer fires (undergoes a randomizer cycle).

Note that if you choose **neither onStart nor f?** for a randomizer, it will **automatically enable onStart** to ensure that it runs at least once.

Putting it all together
Let's begin with the most basic, and most-often used randomizer: A randomizer that at mission start chooses one flag out of a set.

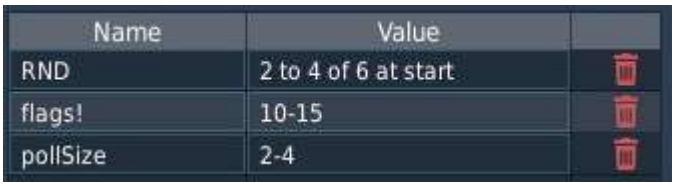


The randomizer on the right creates a set of five flags: "1, 2, 3, 4, 5".

Since there are no other attributes, the following happens when the mission starts:

- No pollSize attribute is present, so pollSize is set to 1 (one)
- No "method" attribute is present, so method is set to "on"
- Neither "onStart" nor "f?" attributes are given, which causes onStart to be automatically set to true
- Since onStart is true, fire! Is scheduled for 0.25 seconds after mission start
- At 0.25 seconds mission time, the randomizer fires
  - A flag from the set "1, 2, 3, 4, 5" is chosen
  - The chosen flag is set to 1 (method = "on")

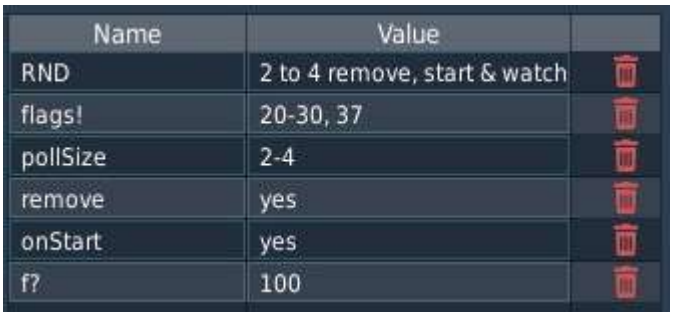
So let's get a bit more fancy:
we have a mission with 6 groups of enemies. We want that two to four of them activate at mission start. Each group is activated by their own flag, numbered 10 through 15



We add two additional attributes: pollSize and rndPollSize. Let's see what happens when we run the mission:

- No "method" attribute is present, so method is set to "on"
- Neither "onStart" nor "f?" attributes are given, which causes onStart to be automatically set to true
- Since onStart is true, fire! Is scheduled for 0.25 seconds after mission start
- At 0.25 seconds mission time, the randomizer fires
    - pollSize is set to a random number between 2 and 4, including 2 and 4. Let's assume 2 is chosen
    - Two flags from the set "10, 11, 12, 13, 14, 15" are chosen
    - The two chosen flags are set to 1 (method = "on"), which activates the groups that are connected to the two flags

Now let's go wild. We have a number of enemy groups, each activated by their own flag. They are activated by flags in the range from 20-30, and another group at flag 37.



We use flag 100 to activate the randomizer, and each time we run a cycle, we want to activate 2 to 4 of the groups. When the mission starts, the first batch of enemies should activate. Also, when a group has been activated by the randomizer, it should not be activated again in later cycles.

So let's dry-run this setup to see what happens when the mission is run:

- No "method" attribute is present, so method is set to "on"
- Since onStart is true, fire! Is scheduled for 0.25 seconds after mission start
- This randomizer watches flag 100 for changes
- At 0.25 seconds mission time, the randomizer fires
    - pollSize is randomized to a value from 2 to 4. Let us assume that "3" is chosen
    - Three flags from the set "20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 37" are chosen. Let us assume they are "26, 21, 29"
    - The three chosen flags are set to 1 (method = "on"), which activates the groups that are connected to these flags
    - The three items "26, 21, 29" are removed from the set for this randomizer
- (some time later) flag 100 changes, causing this randomizer to fire:
    - pollSize is randomized to a value from 2 to 4. Let us assume that "2" is chosen
    - Two flags from the (reduced) set "20, 22, 23, 24, 25, 27, 28, 30, 37" are chosen. Let us assume it's "28, 20"

    - The chosen flags “20” and “28” are set to 1 (method = “on”), which activates the groups that are connected to this flag
    - The items “20” and “28” are removed from the set for this randomizer
- (this repeats until all items are removed from the set)
- (a little later yet again) flag 100 changes, causing this randomizer to fire
    - The randomizer notices it has no items in its set and simply does nothing.

Further Considerations
rndFlags allows you to quickly add the ability to change several randomly selected flags from a set of flags. On a purely ME level, this requires that you then also add triggers to your mission that watch these flags, and perform an action (like, for example, activate a group) when a flag changes.

Note that many DML modules (those who support query ‘?’ attributes) watch flags by themselves and initiate action without requiring ME triggers. You can quickly write a highly randomized mission by placing a bunch of Spawn Zones and wire their query flags to those that are changed by a randomizer. One of the most powerful mission design concepts is combining a randomizer (rndFlags) with a pulser (pulseFlags) and Clone Zones to quickly create an unpredictable, near-infinite range of mission set-ups.

ME INTEGRATION
This module is all about ME integration. It uses Zones only to be able to feed attributes into the module, and then fires the ME flags as described, so you can place the zones wherever you please. To control a randomizer with a flag, use the f? attribute

Name	Value	Description
f? in? rndPoll?	Name	DML Watchflag to start a new random cycle. Defaults to <none set> You can use any synonym, but only one per Zone.
flags!	List of numbers and ranges	Defines the set of flag names that the randomizer chooses the flags from and then sets them according to the “method” attribute.
done+1	Name	The flag name to increase when the randomizer has run out of flags to change, and reshuffle is false (randomizer did nothing) Is increased every time that the randomizer runs a cycle on an empty flag set

Supports DML Flags

3.4.3.2 Dependencies
rndFlags requires dcsCommon and cfxZones

3.4.3.3 Module Configuration
To configure rndFlags via a configuration zone,

- Place a Trigger Zone in ME anywhere

- Name it “rndFlagsConfig” (note: name must match exactly)
- Add any of the following attributes to this zone:

Name	Description
verbose	Show debugging information. Default is off

3.4.3.4 ME Attributes

Name	Description
RND	Marks this as a randomizer. The value of this attribute is ignored **MANDATORY**
flags!	Set of flags, as a comma (‘,’) separated list of the flag names that can be chosen from. The flag names can appear in any sequence. Supports ranges like “2-7” (ME numbers only). Flag names can be included multiple times, including the same flag name multiple times simply increases the likelihood that this number is chosen. Examples: “2, 4, A, A, F, 6” “A9, 3-18, C33, samAttack, 11-11” **MANDATORY**
method rndMethod	DML Output flag method. **Defaults to “on”**
pollSize	Number of items to choose from the set of flags during a cycle. Can be a range: two numbers separated by a hyphen, e.g. “2-5”. When a range is given, pollSize is randomized each cycle to a number between the lower and upper bounds, inclusive. Defaults to 1.
remove	When set to true, the flags that were chosen during a cycle are removed from the set of flags. Defaults to false.
reshuffle	When set, the original full set of flags is restored when all flags have been removed Defaults to false
f? in? rndPoll?	DML Watchflag to start a random cycle. Defaults to <none set> You can use any synonym, but only one per Zone
triggerMethod rndTriggerMethod	Watchflag condition when to trigger. Defaults to “change”
onStart	If true, a cycle is run for this randomizer 0.25 seconds after the mission starts. Defaults to false **NOTE**: if no f? (or synonym) is specified, and onStart is false, onStart is automatically set to true so the randomizer runs once.
done+1	The flag number to increase when the randomizer has run out of flags to change, and reshuffle is false (randomizer did nothing) Is increased every time that the randomizer runs a cycle on an empty flag set

Supports DML Flags
Supports DML Watchflags

3.4.3.5 API

None.

3.4.3.6 Using the module

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

Add a zone to the mission and add the ‘RND’ and “flags” attributes to that zone.

3.4.4 pulseFlags

3.4.4.1 Description

This is the second essential ME flag manipulation module: while rndFlags allows you to randomly select many from many flags and set them, pulseFlags allows you to change the same flag multiple times, over time.

Use pulseFlags to drive repetitive actions and sequences in your missions.

"Pulsing"

In the context of DML, "pulsing a flag" means changing the value of a flag one or more times. The pulseFlag module gives mission designers the ability to create 'pulsers' that automatically change a flag in certain ways, one time or many times. The pulser has the ability to randomize the time between pulses.

A pulser always works with a flag that the pulser changes, a method how to change the flag, an interval (time) between changes, and an optional limit on the number of pulses (changes).

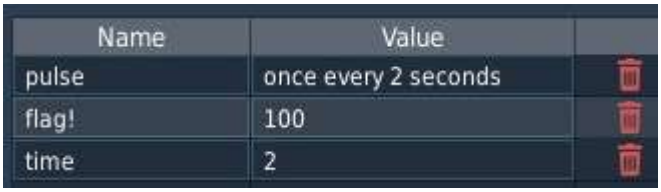
Pulse Number and Frequency

A pulser can change the flag indefinitely, or a set number of times (e.g. 12 pulses). If you set no limit to the number of 'pulses' (flag changes), it runs indefinitely. If you set a maximum number of pulses, the pulser stops after the last pulse (goes into paused state)

Between each 'pulse' (flag change), the pulser waits some time. You can either set this interval time to a fixed number or a range of seconds. When you pass a range, the time between pulses is randomized to fall inside that range (e.g. 3-5 can be any number between 3 and 5, inclusive).

Putting it together

As before, let's start with the bare necessities: create a pulser that indefinitely changes flag 100 once every two seconds.



When run, the pulser on the right:

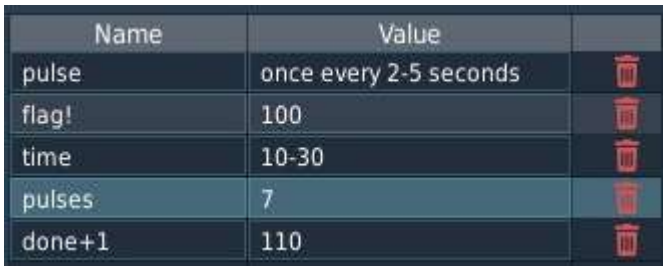
- Sets method to 'flip' (default)
- When the mission starts, flag 100 is 'flipped' (anything other than zero to zero, zero to one) [note that to be precise, the initial pulse starts one second after the mission starts]
- The pulser then schedules the next change to flag 100 in 2 seconds. This repeats until the mission ends or the pulser is paused

Let's make this more random: we want the time between the pulses to vary from 2 to 5 seconds.



As you can see, specifying a range automatically makes the pulser to pick a random number between 2 and 5 (which includes 2 and 5).

Now, lets create a pulser that changes flag 100 exactly 7 times, with 10 to 30 seconds between each pulse, and then increment flag 110 when it is done



Starting a pulser
When the mission starts, pulseFlags automatically starts all pulsers that do not have a paused = true attribute after one second. The one second delay is to allow other modules to load and settle. A paused pulser can be started with the activate? flag.

ME Integration
Like rndFlag, pulseFlag provides comprehensive ME integration to drive most of your flag needs.

Name	Value	Description
pulse pulse!	Name	This is the flag that gets changed according to method each time a pulse is sent.
~~done+1~~ done! pulsesDone!	Name	Increment this flag's value when the number of pulses as defined with the 'pulses' attribute have completed. Note that a pulser without a pulses attribute won't do this.
activate? startPulse?	Name	Watchflag that restarts a paused pulser (with the first pulse)
pause? pausePulse?	Name	Watchflag that when triggered, stops the pulser and puts it into paused state

3.4.4.2 Dependencies
pluseFlags requires dcsCommon and cfxZones

3.4.4.3 Module Configuration
To configure pulseFlags via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is off

3.4.4.4 ME Attributes

Name	Description
pulse **pulse!**	Marks this as a pulser. The value of this describes the flag (number) to change on each pulse. The flag is changed according to the method attribute. Use only one synonym per zone. **MANDATORY**

Name	Description
method pulseMethod	DML flag output method. Use only one synonym per zone. Defaults to “flip”
done! pulsesDone!	This flag’s value is changed when the pulser completes a fully run of pulses. Can only happen when the pulses attribute supplies a positive number. Defaults to <none>
activate? startPulse?	Watchflag. When triggered, a paused pulser is reset and then restarted. Use only one synonym per zone. Defaults to <none>
pause? pausePulse?	Watchflag. When triggered, a pulser is paused. Use only one synonym per zone. Defaults to <none>
paused pulseStopped	When true, the pulser does not start at the beginning of a mission but waits for a trigger on activate?-flag. Defaults to false
pulses	The number of pulses to complete. If set to -1, the pulser runs until the mission ends or the pause?-flag changes Defaults to -1 (endless)
time	Seconds between pulses. You can supply a range (two numbers separated by a hyphen, e.g. “4-19”), the time between pulses is randomized after each pulse to a number in that range. Defaults to 1
zeroPulse	Usually, a pulser starts with an initial pulse (“pulse zero”). This initial pulse can be delayed by *time* by setting zeroPulse to false The effect is that the initial pulse happens after the first delay Default is true (initial pulse immediately)

Supports DML Flags

3.4.4.5 API

None

3.4.4.6 Using the module

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

Add a zone to the mission and add the ‘pulse’ and “flag!” attributes to that zone.

3.4.5 delayFlags

3.4.5.1 Description

A simpler version of pulseFlags with a simpler premise: the zone watches for a flag to change, and a certain later, it changes a different flag. The delay can be randomized.

ME INTEGRATION

delayFlag is intended to introduce a controllable / randomizable delay between the input signal and output signal. It's therefore designed to primarily stack on zones with minimal requirements. Use DML zone-local flags for easy zone integration and replication.

INPUT/OUTPUT

The input functions like a combined delayed trigger and dead-man switch. Once activated, a timer runs down. When the timer has run down, the output flag is changed. This is the basic delay function.

If the input line changes while the timer is running down, the timer is re-started. This is a dead man switch setting: as long as the input changed before the timer runs down, the output is not changed.

If the input line changes after the timer has run down, it simply restarts the cycle (with dead-man switch ability)

RANDOMIZER

You can give a delay range. Each time the start flag is changed, a new delay is picked at random from that range.

3.4.5.2 Dependencies

dcsCommon, cfxZones

3.4.5.3 Module Configuration

To configure delayFlags via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is off

3.4.5.4 ME Attributes

Name	Description
timeDelay	Marks this as a delayFlag module. The value of this attribute defines the number of seconds to wait after activation before the output flag is set. Value can be a range in which case delayFlag picks a random number inside the range (including bounds). Defaults to 1 second **MANDATORY**
out! delayDone!	The flag to bang! after the delay has passed. Use onlyone synonym per zone
method delayMethod	DML method for output Defaults to “inc”
f? in? startDelay?	Watchflag that starts the delay. Use any synonym, but only one per zone.
triggerMethod delayTriggerMethod	DML Method for Watchflags
stopDelay?	Watchflag that stops a running delay. Has no effect on a stopped delay

Supports DML Flags
Supports Watchflags
Supports zone-local verbosity

3.4.5.5 API

3.4.5.5.1 delayFlag.startDelay(theZone)

Starts the delay on theZone. If the delay is still counting down, it is refreshed.

3.4.5.6 Using the module

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

Add a zone to the mission and add the ‘timeDelay’ and “out!” attributes to that zone.

3.4.6 Raise Flag

3.4.6.1 Description

This is the DML version of ME's ability to set flags. It allows you to set flags directly, without a trip to ME's trigger panel. RaiseFlag has full support for DML flags. In conjunction with local flags (which are supported), raiseFlag becomes a more versatile zone-local replacement for many module's onStart input (which it replaces while we deprecate onStart for most modules)

A raiseFlag module can increment, decrement, flip, or set a flag to a specific value some time after the mission starts. You can randomize the time, and you can prevent it from setting the flag if you stop if before its timer runs down by using the stopFlag? input. The latter allows you to easily design 'race against clock' scenarios.

Use this to set flags in your mission to specific values – either at the start of the mission, or after some (random) time.

3.4.6.2 Dependencies

dcsCommon, cfxZones

3.4.6.3 Module Configuration

To configure raiseFlags via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is off

3.4.6.4 ME Attributes

Name	Description
raiseFlag **raiseFlag!**	Marks this as a flag raiser. The value of this attribute is the flag that is to be raised. Use only one synonym per zone **MANDATORY**
value	Method or value to set the flag to. Supports • `'inc'` increment the flag's current value by one • `'dec'` decrements the flag's current value by one • `'flip'` sets the flag's current value to 0 if it was anything but zero, and to 1 if it was zero. • (number) – set the flag to this number Default is 'inc' – the flag's value will be incremented by one

Name	Description
afterTime	Amount of time (in seconds) after mission start to set the flag. Can be a range. If a range is given, the time is a random number from this range. **If this attribute is omitted, the flag is set 0.5 seconds after the mission starts**.
stopFlag?	Only useful in conjunction with afterTime. A Watchflag. When triggered and raiseFlag is still waiting for afterTime, raiseFlag is 'disarmed' and raising the flag is cancelled. It will not be raised later. Once stopped, it cannot be re-started.
triggerMethod raiseTriggerMethod	Watchflag condition for stopFlag?

Supports DML Flags

3.4.6.5 API

Tbc

3.4.6.6 Using the module

3.4.7 xFlags (Flag Combining / Testing)

3.4.7.1 Description

More flag testing than you can shake a signal at. Although it's one of the most sophisticated modules in DML's arsenal, it's easy to use and indispensable when you want some decision-making logic or a way to determine that a certain situation has arisen, but don't want to break out Lua.

Consider:

- you have five enemy groups, and when half or more of them are defeated, you want to call in support.
- The allies have captured two of six possible waypoints, but not yet attacked the main base, so you want to trigger a bomb raid
- Red enemy generator has created five waves, blue is still holding their own base, and all red groups are destroyed, so you want to set the win condition

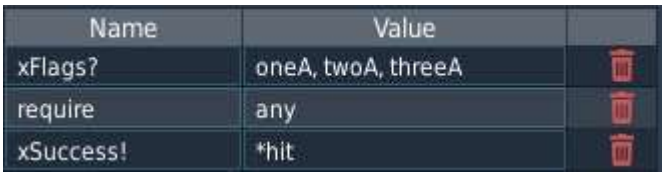
So, essentially, you'll break out xFlags whenever you need some decision testing, to see if enough of a mission's goals are fulfilled to warrant some action.

BASIC SETUP: TESTING AND xSUCCESS!

Essentially, xFlags consists of a set of input flags and the requirement that a number of them must have triggered. **Once that requirement is fulfilled, xFlags triggers the output xSuccess and pauses until reset.**

Let's assume you have three input flags (see below): A, B and C. You require that at least two of them need to trigger. So, after *any combination* of two flags trigger (AB, BA, AC, CA, BC or CB), the output flag is triggered. xFlags gives you fine control over the requirements that must be met, so you can quickly build a zone that decides win conditions, trigger new stages or calls in support.

INPUTS (FLAGS)



xFlags uses a list of input flags ("xFlags?") that is inspects once per cycle (usually once every second) to determine if they fulfil the input's "xFlagMethod" watchflag trigger condition. In other words, it checks and internally stores which input flags have triggered. The result of each individual check is saved and later checked at xFlag's "require" attribute (see below). The "xFlagMethod" attribute follows standard Watchflag Method conventions for input flags; it applies to all input flags listed in the "xFlags?" attribute, and defaults to 'change', meaning that by default an input's value is compared against the "zero state" (see below) to determine if it triggers.

At mission start (and upon triggering the 'xReset' input) xFlags sample the current state of all input flags and save it as a "zero-state" reference (see below) so they can compare against this to determine if a flag has changed.

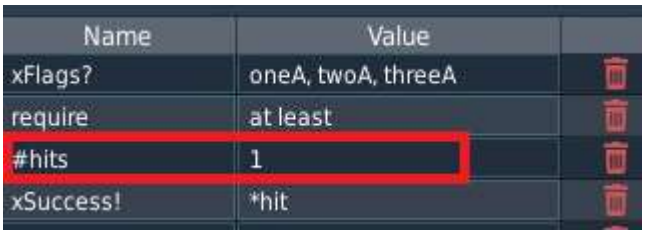
REQUIREMENT
The most important part in xFlag's inspection cycle comes after it has established all input flag's trigger state: it then compares how many input flags currently have triggered against the requirement of how many should. If the requirement is fulfilled, the xSuccess output is banged according to "xMethod".

Currently, xFlags understands the following conditions:

- 'or', 'any', or 'some'
  triggers if at least one of the input flags has triggered. This is equivalent to the logical 'OR' operation performed over all flags (hence one of its synonyms is 'or'). **This is xFlag's default requirement.**

- 'and' or 'all'
  triggers if all the input flags have triggered. This is equivalent to the logical 'AND' operation performed over all input flags (hence the synonym 'and')

- 'more than'
  triggers if more input flags have triggers than the number given in the '#hits' attribute



- 'at least'
  triggers if #hits or more of the input flags have triggered

- 'exactly'
  if triggers if the number of input flags that have triggered is equal to #hits

- 'none'
  triggers if none of the input flags have triggered. Requires that you turn off one-shot mode, as the initial setting (all flags set to false) will trigger with 'none', and immediately stop the xFlag

- 'not all' or 'nand'
  triggers when not all input flags have triggered. Requires that you turn off one-shot mode, as the initial setting (all flags set to false) will trigger with 'none', and immediately stop the xFlag. This is equivalent to the logical 'NAND' operation performed over all inputs.

- 'most'
  triggers when more than half of the input flags have triggered. Will not trigger is exactly have have triggered, so be careful if the number of input flags is even.

- 'half or more'
  triggers when half or more of the input flags have triggered. Will also trigger if exactly half of all flags have triggered

- 'never'
  used when you are using xFlag's "direct outputs" (xDirect, xCount, xChange) and

want it to operate during the entire mission. xFlags will never trigger xSuccess, and keep evaluating, setting xDirect, xCount and xChange accordingly

"LATCHED" ZERO STATE AND RESET?

When the mission starts, all xFlag zones "reset": the save their initial state (i.e. save the state of the input flag at that moment into a 'zero state'). Each time xFlag re-examines the flags (usually once per minute), it determines each input's current state against that flag's zero state to arrive at the trigger status. It continues to do so until the main requirement has been fulfilled, at which point that xFlags zone stops evaluating until it is reset.

When you reset an xFlag (implicitly at mission start, explicitly with the xReset? input), it will save all current input states as the new state zero, and re-start evaluating. An xFlag that is still running (i.e. not all requirements have yet been met) *can* be reset.

DIRECT OUTPUT: xDIRECT, xCHANGE! and xCOUNT

Many uses of xFlag involve constant evaluation of the input flags without necessarily requiring there to be xSuccess signal. For example, you might want a xFlags to constantly count how many of the input flags are currently set to a value equal to two, perhaps to be used in another module. xFlags provides additional outputs that are constantly (once per cycle) updated with results from the evaluation process that you can use:

- *xChange!*
  A DML flag that is banged whenever xFlags detects a change from last to current state (at least one result of the flags is different from the last check)

- *xDirect*
  The current raw result of xFlag's entire evaluation, either 0 (false) or 1 (success). Useful when one-shot is turned off (see below). Note that this is very different from xSuccess, as xSuccess is governed by a DML method, and triggers only on a change.

- *xCount*
  Is always set to the number of successful input flag tests. If, for example, three of the input flags test successfully in this cycle, the value of this flag is set to the number 3.

If you are primarily interested in the direct outputs remember to set "require" to "never" to ensure that xFlags will always continue to evaluate the flags (require defaults to "some", so unless you change that, xFlags ceases to evaluate after the first hit)

ONE-SHOT

Unless you reset an xFlag, it runs only until it triggers, and then pauses. You can change this behavior by setting the oneShot attribute to false. In this case, the xFlag continues to evaluate all inputs every cycle (usually once per second), and trigger the output according to the requirement fulfilment.

SINGLE-INPUT xFLAGS (FLAG CONVERSION)

It will have occurred to you that you can use xFlags in a particularly useful way: to convert a flag's value into a trigger by simply providing a single input flag with the relevant xFlagMethod.

That is indeed possible and works well. You may be better served if you look to the "changer" module which provides a similar service in a neater package.

Since xFlags' functionality can be suspended and enabled by flags, it can function as 'gated switch', a device that only permits signals to pass when it is enabled. Again, the changer module may be a better choice.

SUSPENDING XFLAGS: xON?, xOFF? And xSuspend

Often, it is desirable to suspend an xFlag. Use the xOn? and xOff? inputs (hard-wired to a change method) to enable and suspend an xFlag. Usually, an xFlag starts the mission in enabled state. When you add an xSuspend attribute with the value true, the xFlag starts in suspended state and will only start once it receives a change on the xOn? input.

Note that even while suspended, xFlags do receive and respond to xReset? by loading a ne zero state

LIMITATIONS AND CAVEATS

"Change" and "flip"

The default 'change' watchflag trigger condition watches for a change on a flag, and when the value of that flag changes with regards to the previously read value, it triggers. During an xFlag cycle, that process is slightly different: the current value is compared to the value it had when the xFlag was last reset: the "zero state". This has a number of important implications when xFlag is repeatedly checking all flags during cycles:

- If an input flag is changed multiple times, and the end result is still different from the initial "zero state" value, only one trigger event is registered
- If an input flag changes multiple times, and the newest value happens to be the same as the one stored in the "zero state" (for example when using the 'flip' trigger method), this registers as *not triggered*, irrespective of how many times it changed in the mean time

Using requirement 'most'

Also be careful when using the 'most' requirement – it will only trigger the xFlag's output when more than half of the input flags have triggered. This is important to remember when you have an even low number of flags. For example, if you have 4 (four) input flags, 'most' means that 3 (three) or more flags must have triggered in order to fulfill then requirement.

It takes time…

Remember that xFlags have a cycle time (usually one second). If you build a cascade of xFlags (and you will), remember that it may take some time for a signal to propagate through multiple levels, so make sure that you wait a couple of cycles before you suspect an error in your setup. Use the zone's 'verbose = yes' attribute to see which inputs an xFlag receives when. xFlags understands zone-local verbose attributes.

3.4.7.2 Dependencies
xFlags requires dcsCommon and cfxZones.

3.4.7.3 Module Configuration
To configure xFlags via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is off
ups	Number of cycles per second. Default is 1

3.4.7.4 ME Attributes

Name	Description
xFlags?	A list (comma-separated) of input flags whose values should be evaluated to form the output signal **MANDATORY**
require	Condition/Operation that should apply to the input flags to form the output value. Currently supports the following conditions: • ‘or’, ‘any’, or ‘some’ triggers if at least one of the input flags has triggered • ‘and’ or ‘all’ triggers if all the input flags have triggered • ‘more than’ triggers if more than the value given in ‘#hits’ of input flags have triggered • ‘at least’ triggers if #hits or more of the input flags have triggered • ‘exactly’ if triggers if the number of input flags that have triggered is equal to #hits • ‘none’ triggers if none of the input flags have triggered. Requires that you turn off one-shot mode • ‘not all’ or ‘nand’ triggers when not all input flags have triggered. Requires that you turn off one-shot mode

Name	Description
	• ‘most’ triggers when more than half of the input flags have triggered. Will not trigger is exactly have have triggered, so be careful if the number of input flags is even. • ‘half or more’ triggers when half or more of the input flags have triggered. Will also trigger if exactly half of all flags have triggered • ‘never’ used when you are using xFlag’s “direct outputs” (xDirect, xCount, xChange) and want it to operate during the entire mission. xFlags will never trigger xSuccess, and keep evaluating, setting xDirect, xCount and xChange accordingly Defaults to ‘some’
#hits	Value used for only some of the require attribute. Can be a value or flag name (in which case the value will be loaded from that flag). Numbered flags must be enclosed in double quotes, e.g. “22” to access flag 22. Defaults to 1 (one)
xFlagMethod	Condition that must be met for individual input flags. Is identical to trigger method for Watchflags except it is applied to each input flag individually. Defaults to “change”
xSuccess! out!	Flag to bang! when xFlags when the evaluation of input flags succeeds (all conditions are met). Once xSuccess is triggered, and unless oneShot is set to false, this zone’s xFlag pauses until xReset is triggered. Defaults to <none>
xChange!	Flag to bang! when xFlags detects a change in the input configuration. Merely detects a change in the input configuration, has no relation with xSuccess!, except that xSuccess will also be accompanied by a bang on xChange! Defaults to <none>
xDirect	Each time xFlags evaluates the input flags, it directly sets the xDirect flag to the evaluation result (0 or 1) – this is different from what xSuccess may output, since that flag’s value is dependent on the xMethod attribute. **This flags value is set directly, not via DML method.** Defaults to <none>
xCount	Each time xFlags evaluates the input flags, it directly sets the xDirect flag to the number of hits (positive test results from the individual flags tests). For example, if three tests of the input flags are successful, xFlags sets the value of this output to the number 3 **This flags value is set directly, not via DML method.** Defaults to <none>
xReset?	When the value of this input changes, the zone’s xFlag module is reset, and evaluation starts afresh.

Name	Description
	Note: this input always reacts to a change in the flag's value Defaults to <none>
method xMethod	DML method for output flags Defaults to "inc"
oneShot	When the value if this attribute is false, that zone's xFlag module will not stop evaluating after it triggers xSuccess.
xOff?	Flag to turn the xFlag off, suspending it. When turned off, no processing of input flag occurs. The xFlag will still respond to xReset by loading a new zero state. **Note: this input always reacts to a change in the flag's value** Defaults to <none>
xOn?	Turns a suspended xFlag back on to resume processing. It resumes processing where it left off **Note: this input always reacts to a change in the flag's value** Defaults to <none>
xSuspended	Sets the initial state of xFlag. Setting it to true suspends the xFlag at mission start Defaults to false

Supports DML Flags
Supports DML Watchflags
Supports zone-local verbose

3.4.7.5 API

(tbc)

3.4.7.6 Using the module

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

To use, simply add the 'xFlags? and required other attributes to a zone.

3.4.8 countDown

3.4.8.1 Description

Whenever you need to know when a certain event occurred a certain number of times, you can look to countdown to help you. There are surprisingly many uses for this: delaying actions or responses, filtering every other event, limiting the number of respawns, counting kills towards victory etc. You'll find that in almost every mission you need counters, and that is why ME provides a (limited) support of counting on flags as well.

So – big picture: countDown is an ME flag monitoring and manipulating module that counts for you.

BASIC FUNCTION

countDown is easy to understand: it has an input flag that it watches (the 'in?' flag), and you set a limit (e.g. 5). After countDown has seen that number of value changes on the 'in?' flag, it changes the value of 'out!' flag. A simple count down.

ADDITIONAL FEATURES

While a counter is always helpful in mission design, it's the built-in decision making that makes countDown so helpful. countDown offers additional flag handling outputs that make

- **T-Minus (Output)**
  Named after NASA's famous count down procedure, the "tMinus!" output simply changes each time a signal is received on the "in?" flag, but the count is still above zero. Hence, tMinus produces a change every time before zero is reached, but not when zero is reached.

- **Looping (Attribute))**
  You can set the count down to automatically restart counting when it has reached zero.

- **belowZero (Output)**
  This is the opposite of T-Minus: it fires every time an input is received on 'in?', and the counter is below zero, i.e. the count down has received more signals on the 'in?' flag than the start value specified, and is now 'below zero'. This can't happen if the count down is set to loop

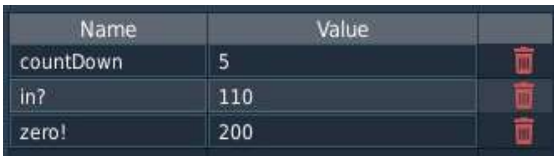
- **Value Randomization (Attribute)**
  The start value can either be given as a number (e.g. 5) or a range (e.g. 3-7). If loop is set to true, a new randomized start value is calculated on each iteration of the loop.

COMMON COUNTER USES / SET-UPS

Counters are one the most versatile components in process design. Here are some of the most common use cases and their configuration that you can use to solve common mission requirements

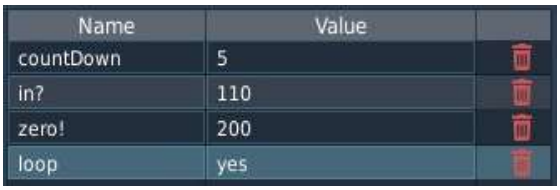
- **Classic Count-Down: ONCE AFTER SO MANY TIMES**

  In this configuration, you simply set the start value, and zero! changes after the correct number of changes were detected on 'in?'



- **EVERY NTH TIME – FREQUENCY DIVIDER**

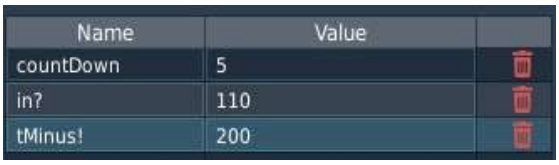
  This is a modification of above: the same set-up, except we have enabled 'loop'. After setting the start value (here 5), we now see a change on zero! every n-th change on in? (here every 5th time)



  This configuration is also called a 'frequency divider', as it divides the frequency of the occurrence of in? by n.

- **ONLY UNTIL**

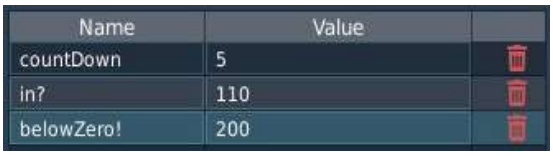
  This configuration uses a count down as a "limiter": it allows a change on 'in?' to propagate to the output 'tMinus!', but only until the maximum number of changes has been received. After that, it no longer transmits changes in 'in?' to the output 'tMinus!'. So, in this configuration, only a maximum number of changes appear on 'tMinus!'.



  Note:
  when you set the start value to n, 'tMinus!' will propagate only (n-1) signals: on the final signal the value is zero and 'out!'/'zero!' fires instead. If you need exactly n signals from the counter, you need to set 'out!' *and* 'tMinus!' to the same flag (or increase the start value by one).

- **EVER AFTER / NOT BEFORE**

  This is the opposite of 'ONLY UNTIL': every time there is a change after the counter has counted to zero, the signal on 'in?' is propagated to 'belowZero!'. As before, you may or may not want the zero change to count as well, so you may also have to set out! to the same flag as 'belowZero!'



ME INTEGRATION

countDown provides comprehensive ME flag integration.

Name	Value	Description
in? count?	Flag Name	Watchflag. When triggered, decreases count by one Defaults to <none>
tMinus!	Flag Name	DML Ouput. Changes when counter decreases and has not yet reached zero Defaults to <none>

Name	Value	Description
zero! out!	Flag Name	DML Output. Changes when counter decreases and reaches exactly zero. Defaults to <none>
belowZero!	Flag Name	DML Flag. Changes every time count is decreased and is below zero. Note that if the 'loop' attribute is set, this can never happen. Defaults to <none>
counterOut!	Flag Name	A flag, when given, is set to the current value of the count. The flag is updated each time that the counter is triggered via count?
disableCounter?	Flag Name	A Watchflag that, when triggered, disables the counter for good.

3.4.8.2 Dependencies

countDown requires dcsCommon and cfxZones

3.4.8.3 Module Configuration

To configure rndFlags via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is off
ups	Number of updates per second

3.4.8.4 ME Attributes

Name	Description
countDown	Marks this as a count down. The value of this attribute defines the number times until the count reaches zero This value supports ranges: if you specify a range (e.g., "3-5") each time the count down is initialized (at start, and when looping), a random number in the range (including upper and lower limit) is chosen. Defaults to 1 (one) **MANDATORY**
loop	If this attribute is true, a count down restarts after reaching zero. If the count down is given as a range, an new random start value is taken from that range (including upper and lower limit)
method ctdwnMethod	DML Flag method for output flags. Use only one synonym per zone. Defaults to "flip"
count? in?	Watchflag that when triggered constitutes a signal to count down by one. Use only one synonym per zone. Defaults to <none>
triggerMethod	Watchflag method for input flags.

Name	Description
ctdwnTriggerMethod	Defaults to “change”
zero! out!	DML Output flag to bang when countdown reaches zero. Use only one synonym per zone. Defaults to <none>
tMinus!	DML Output flag to bang when count is lowered, and has not yet reached zero. Defaults to <none>
belowZero!	DML Flag. Changes every time count is decreased and is below zero. Note that if the ‘loop’ attribute is set, this can never happen. Defaults to <none>
counterOut!	A flag, when given, is set to the current value of the count. The flag is updated each time that the counter is triggered via count? Defaults to <none>
disableCounter?	Watchflag that when triggered turns off the count down. Defaults to ‘<none>’

3.4.8.5 API (tbc)

3.4.8.5.1 getCountDownZoneByName(aName)

3.4.8.5.2 isTriggered(theZone)

3.4.8.6 Using the module

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

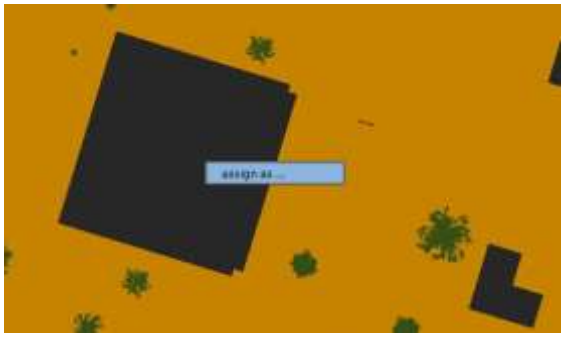
Add a zone to the mission and add the ‘countDown’ attribute to that zone. Add other attributes as required

3.4.9 cfxObjectDestructDetector

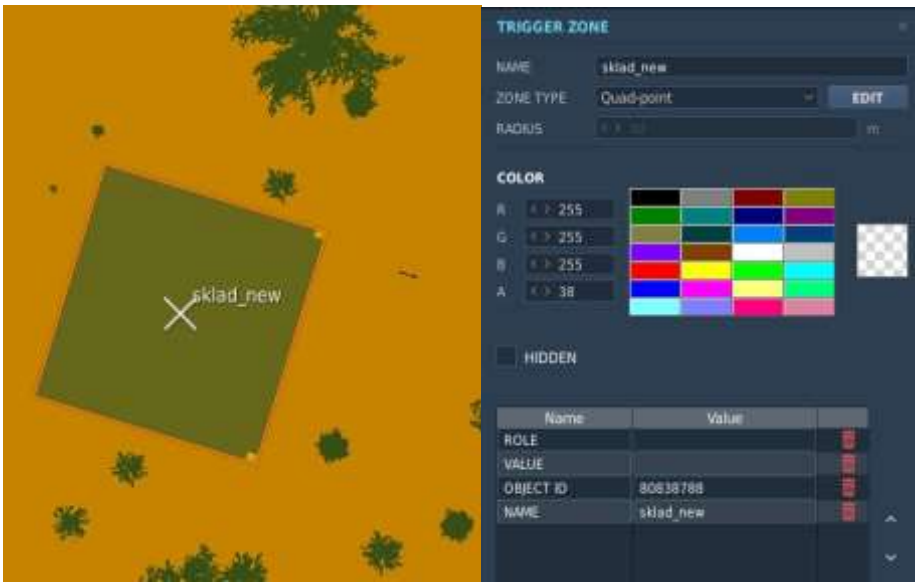
The destruct detector is a module that tightly integrates with ME, and additionally provides a scripting API. Unlike most other zone enhancements, this module provides direct means to manipulate ME-accessible flags.

3.4.9.1 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! destroyed! objectDestroyed!	name	DML flag to bang! when object is destroyed. Use any synonym but not both in the same zone

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.

3.4.9.2 Dependencies
This module requires dcsCommon and cfxZones to be loaded

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

3.4.9.4 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 **MANDATORY**
NAME	THIS ATTRIBUTE IS FILLED BY ME AND MUST NOT BE CHANGED **MANDATORY**
~~f=1~~	Sets the flag specified in Value to 1 when object is destroyed (deprecated – use method and f!)
~~f=0~~	Sets the flag specified in Value to 0 when object is destroyed (deprecated – use method and f!)
~~f+1~~	Increases the value of the flag specified in Value by 1 when object is destroyed (deprecated – use method and f!)
~~f-1~~	Decreases the value of the flag specified in Value by 1 when object is destroyed (deprecated – use method and f!)
method oddMethod	DML Method to bang! on f! when object destroyed Defaults to “flip”
f! destroyed! objectDestroyed!	The flag to bang! when the object is destroyed. Use only one synonym per zone. Defaults to “*none”

3.4.9.5 API

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

3.4.9.5.1 addCallback(theCallback)

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

3.4.9.6 Using the module

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

3.4.10 cfxSpawnZones

3.4.10.1 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, activation and pause)

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? spawn? spawnUnits?	Number	Watches the flag <Number> (as accessed by DCS) for a change. **Each time the flag's value changes, a new group is spawned** Use only one synonym per zone
activate?	Number	Watches the flag <Number>. Each time the flag's value changes, the spawner's 'paused' setting is forced to 'false'. Used to 'activate' a paused spawner
pause?	Number	Watches the flag <Number>. Each time the flag's value changes, the spawner's 'paused' setting is forced to 'true. Used to 'pause' a spawner

This simple mechanism allows mission designers to, for example, spawn troops whenever a player unit enters a zone (for very nasty surprises). Furthermore, spawners support watchflags that allow the mission to pause and un-pause (activate) spawners by changing a

flag. This allows other modules (e.g. rndFlags) to activate paused spawners, or turn them off at will.

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

3.4.10.2 Dependencies
Required: dcsCommon, cfxZones, cfxCommander, cfxGroundTroops

Optional: cfxHeloTroops

3.4.10.3 Module Configuration
This module does not need to be configured

3.4.10.4 ME Attributes

Name	Description
spawner	Marks this ME Zone as a spawn zone. **Value of this attribute is ignored**, use it to describe what it spawns to make mission editor easier for you **MANDATORY**
f? spawn? spawnObjects?	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 Use only one synonym per zone
pause?	Flag to observe. Each time the flag's value changes, the spawner's 'paused' setting is forced to 'true. Used to 'pause a spawner
activate?	Flag to observe. Each time the flag's value changes, the spawner's 'paused' setting is forced to 'false'. Used to 'activate' a paused spawne
types	Type string array for the ground units that are spawned. Example "Roland ADS, Roland Radar, Roland ADS" or "Soldier M4" – **WARNING**: 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 https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB 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 (**Warning**: 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: https://wiki.hoggitworld.com/view/DCS_enum_country).
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

Name	Description
	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. **If provided, baseName MUST BE UNIQUE. If you do not assign a base name, a unique one will be generated for you**. 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
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 it 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 time. 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 `cfxSpawnZones.spawnWithSpawner()`. See the API section on how to get a list of eligible spawners. Automatically interfaces with HeloTroops and other enhancements

3.4.10.5 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.

3.4.10.5.1 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)`

3.4.10.5.2 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 is 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.

3.4.10.5.3 function cfxSpawnZones.getSpawnerForZone(aZone)

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

3.4.10.5.4 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

3.4.10.5.5 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.

3.4.10.6 Using the module

Include the cfxSpawnZones source into a DOSCRIPT action at the start of the mission. Create Spawn Zones by adding a Trigger Zone and adding attributes as described above.

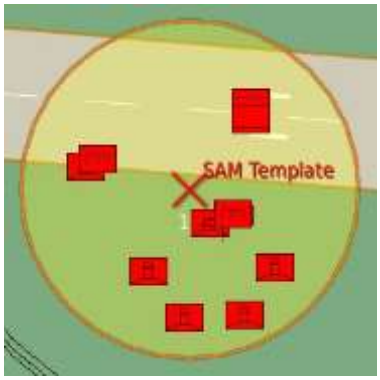
3.4.11 Clone Zones

3.4.11.1 Description

Like Spawn Zones, clone zones dynamically spawn units into the game as the mission runs. Unlike Spawn Zones, you don't use a type string to tell clone zones which units to spawn. Instead, they 'clone' (create exact copies) of 'group templates': a group template is simply a group that exists in the mission. Therefore, creating a template simply consists of creating a group in ME, and then placing a clone zone on top of it. A template is created. Later in the mission, Clone Zones can re-create new groups from these templates.

Creating a Template

This step is quite simple. Whenever you have assembled a nice composition of units (they don't necessarily have to belong to the same group) that you want to use as a template, simply place a clone zone over them (over at least one of the group's units).



At mission start, **all groups who have at least one unit inside a clone zone become part of that zone's clone template**; the Clone Zone then creates a 'snapshot' (an exact copy) – including all route information – of all units and stores it in memory as a template. The template *can then be used by other clone zones as well*.

IMPORTANT

All templates are created at mission start. It is irrelevant what happens to those units later. When the clones are created, they are always fresh copies from the template at mission start.

Creating Clones from "Templates"

Clone zones are used to spawn units. When told to spawn, they take a template, and re-create the entire template around the zone's center from the snapshot that was takes at mission start.



What makes clone zones so powerful is that clone zones **can use other clone zones's templates**, and they can even choose randomly between templates. Using cloners and 'foreign templates' can rapidly accelerate mission design and radically cut down unit proliferation (you clone templates instead of deploying individual units). A common mission design pattern is to create one complex template (e.g. a SAM site), and use multiple clone zones throughout the map that all reference that same template. Since clone zones spawn when told to, using a Flag Randomizer allows you to quickly create a randomized, unpredictable mission with very little effort and only a few templates

Important Note

After a clone zone creates a template from the groups inside, all groups are removed from

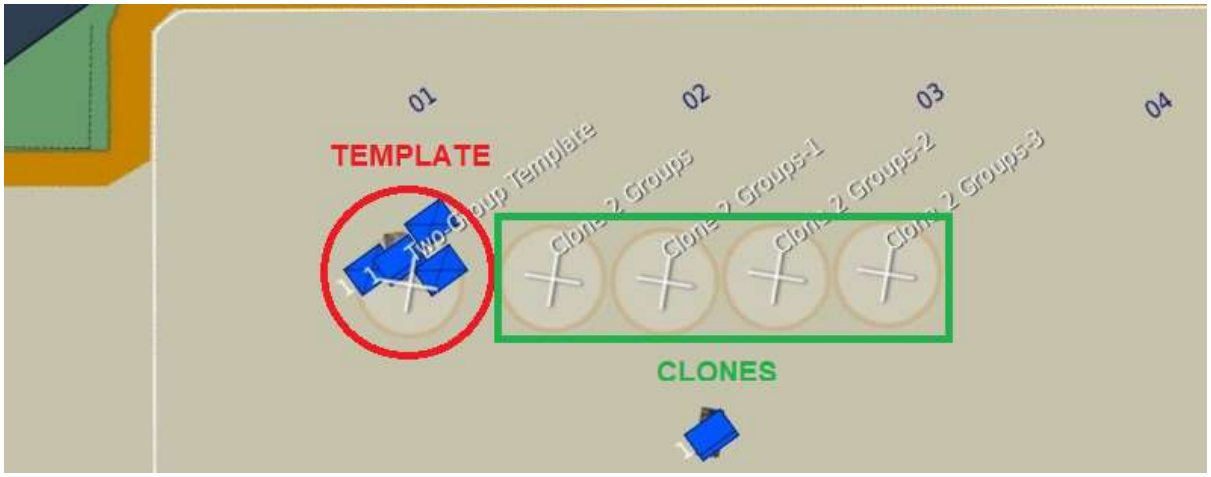
the game as part of the template creation process, and then clones are spawned from the new when the mission starts. You can prevent the template from spawning by adding an 'onStart = false' attribute.

Clone Naming

When a cloner spawns a new group, that clone's name is derived from the original group's name by taking the original name and adding a unique string. For example, if a group in a template is named "Dock Defenders", clones are named using "Dock defenders" as base, and have a new unique string appended. The result may be something like "Dock Defenders-912223". This scheme allows you to easily incorporate cloned groups into unit zones and other DML modules that accept "wildcard" names.

A Quick Example

Let's walk through an example (also part of the 'Attack of the CloneZ.miz').



On the map above, there are two blue groups inside the red circle: One consisting of three Infantry, and one group that consists of a single Hummer. Both groups have at least one unit inside the "Two Group Template" Trigger Zone (marked **TEMPLATE** in the image above), making them part of the template. (Disregard the sole blue unit in front of the green rectangle, that is just eye candy)



"Two Group Template" is set up as a unit template. Hence all groups that have at least one unit inside the zone become part of that template. **To make a clone zone serve as a template, omit the 'source' attribute**. That's all that is required to create a template.

We also note that there is neither a 'spawn?' nor 'onStart' attribute, meaning that we expect this template not spawn any clones, and therefore be empty when the mission starts. Usually, we would add an "onStart" = true attribute on the template as well, but here we want to demonstrate that you can define templates without having the template's units spawn in the mission. This is a convenient way to define complex templates somewhere out of the way without taking up memory and performance.

We now turn our attention to the four identical “Clone 2 Groups” clone zones (marked **CLONES** in the image above).

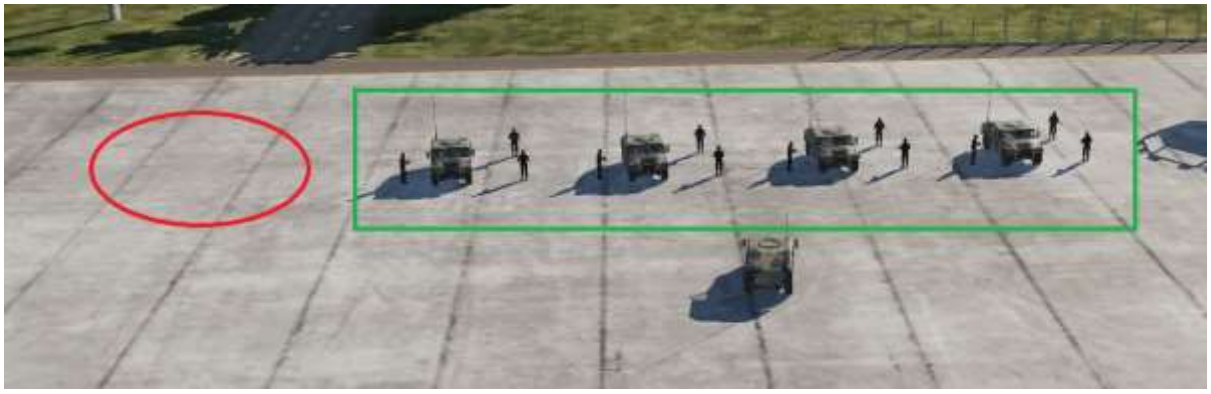
Name	Value	
cloner		
source	Two-Group Template	
onStart	yes	

They all use a “source” attribute, meaning that they use a foreign template, in this case one that is supplied by clone zone “Two-Group Template” – the one we just analyzed. We also note that these zones all have an ‘onStart = true” attribute, meaning that when the mission starts, these zones all first fetch the template from “Two-Group Template”, and spawn a group that looks exactly like the one we assembled in ME for the template zone.

So, when the mission runs, we should expect:

- No units in the red area
- A total of 4 copies of what is in the red area in ME inside the green area

Let’s run the mission:



Nice.

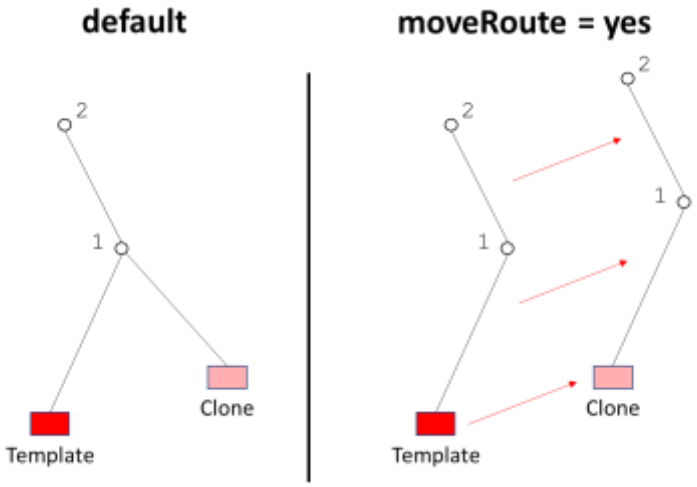
A word of caution

Cloners care little about what they clone and where to – they simply do your bidding. You can tell a cloner to import and spawn any template from anywhere, and you therefore can spawn units in locations where they really should not spawn. While this may seem fun at first, it can get old fast and ruin an otherwise good mission, so use discretion when employing similar tricks like the one below:



Cloning Routes

Since clone zone create exact copies, it happily also copies all route and waypoint information into the template. This can have some unforeseen effects when a different clone zone than the one that created template creates a clone based on that template. When the cloned units are moved to the new clone location, what should happen to the waypoints? Should the also move to the new location? Sometimes, you want multiple clone zone spawn units that all rally to the same point (the first waypoint after the initial point), sometimes you want the route that the units are to follow also move with the unit.

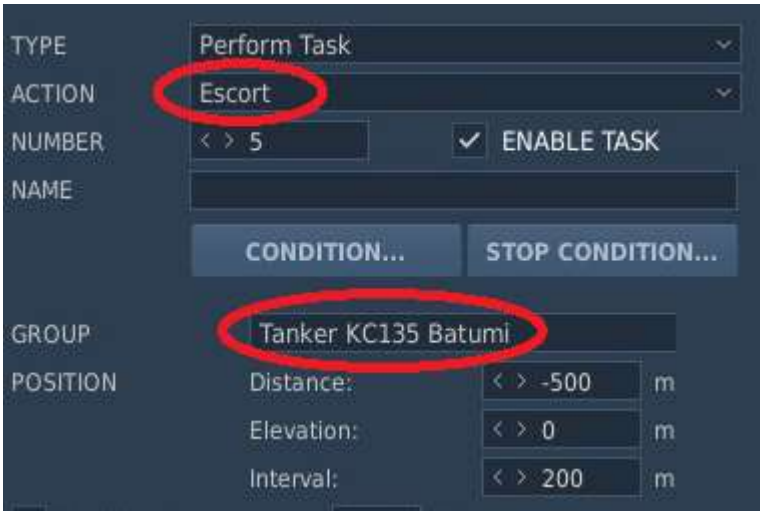


To allow you to do both, Clone Zones treat routes as follows:

- The initial waypoint always coincides with the spawn location of the cloned unit.
- Unless you set the clone zone's moveRoute attribute to true (yes), all waypoints beyond the initial point remain at the original (template's) position. This means that except for the spawn location, all units that spawn from such a cloner's position follow the same route
- If a spawner's moveRoute is set to true, all waypoints for the group receive the same offset as the spawner has from the original template's zone. All units from such a spawner follow a route that has the same shape as the original template's route, but is moved to the spawner's position

Clones References

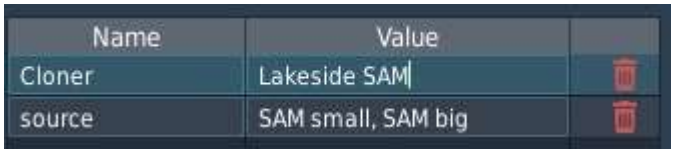
Some of the waypoint actions can reference other groups or units. For example, you can tell a group to escort or follow another group, or attack a unit. So what happens if you clone a group? There are a couple of possibilities:



- If the group/unit the clone is referring to is not a clone, nothing changes. If you, for example have a rescue helicopter that respawns every hour (to get around a much more complex waypoint and order management conundrum) with orders to follow a carrier, the new clone simply also follows carrier
- If the group/unit the clone refers to is a clone, it gets a bit more involved:
    - If the other unit referred to is part of the same spawn template (since spawners support multiple groups per spawn), DML automatically resolves the reference to that other clone
    - If the other unit referred to is par of a different cloner, the group/unit reference is always resolved to the last (most recent) clone of that unit/group.
- The same rules apply to cloned static objects.

Using other zone's ('foreign') and multiple templates

Arguably one of clone zone's most powerful abilities is that they can use other clone zone's templates with their 'source' attribute. You use this to assemble a group (or more) into one template, and can then deploy clones of that template to many other zones throughout the map. A common use case for this is to create a SAM site template and place multiple clone zones that reference that SAM template. At mission start, you randomly (e.g. using rndFlag) activate only some of them, making the mission unpredictable. This dramatically decreases unit count in ME and the effort to create, and maintain, a mission.



To further enhance randomization, clone zones natively support template randomization. When a clone zone supplies more than one template zones for cloning in their 'source' attribute, each time the zone undergoes a clone cycle, it randomly chooses one of the supplied templates to clone.

Aircraft, Helicopters, Ships, Static Objects

Note that clone zones are not limited to ground units. They copy all units (except player aircraft) into their template. That way you can easily create templates for aircraft that attack certain targets, and later spawn them multiple times.

Aircraft
When you clone aircraft groups to other locations, remember that their routes are also cloned, and these routes are changed according to the value of *moveRoute*. Some special rules apply to aircraft routes: if the first waypoint references an airdrome (because the aircraft departs from that airfield: hot/cold start, runway), the closest airdrome to the new position (clone zone's center) is used. The same applies to the route's last waypoint: if the plane is instructed to land at an airdrome, the closest airdrome to the last waypoint is used. The latter is only important when moveRoute is set to true.

Helicopters
Similar restrictions with regards to airdromes apply to helicopters as they do to aircraft. Helicopters that start from the ground may and may not have these restrictions, depending on the helicopter.

Ships
Be careful when cloning ships, as cloners do not check if the position the units are cloned to are eligible for ships. You can clone a ship template and position an oil tanker on a mountain top.

Static Objects
When you include static objects into a template, remember that a spawner's 'empty+1' flag is only triggered if all units are destroyed – and this includes any static objects spawned.

Coalition of spawned clones
By default, a clone all spawned units belong to the same faction as the units that were used to create the template (this means that in extreme cases, the same template can hold multiple coalitions). You can, however, override by specifying a masterOwner zone. All spawned clones then belong to the coalition that owns masterOwner.

Mission Start
When a mission starts, all clone zones run through the following steps:

- If no 'source' attribute is present, the clone zones looks for all units (air, ground, sea) inside the zone, and copies all groups for which at least one unit is inside the zone into a new template. All groups that are part of the template are removed from the game. If no groups are found, a warning is given that no template was created.
- Only if there is an 'onStart = true' attribute present, a clone cycle is initiated.
- Each time a clone cycle starts,
    - If a 'source' attribute is found, a clone zone with the name as gives as value for source is fetched, and that template is loaded. If more than one source templates are given, one is selected by random.
      If that clone zone has no template, a warning is given. Only if no source attribute is given, the clone zone's own template is loaded.

- The template is used to create an exact clone of all groups that are in the template, including all routes. During the cloning process, all unit and group names are made unique based on the original (template's) names.

Since all template's original groups are all destroyed during the template creation process, any units or groups that are part of a template cannot be used in ME's 'UNIT IN ZONE' or 'GROUP IN ZONE' conditions because they no longer exist – they were destroyed during cloning.

ME INTEGRATION

Cloners can spawn on mission start (default), and when a flag changes it's value to signal to the cloner that a new clone cycle should be initiated. Also, the cloner can watch the groups that it has cloned, and change a flag when all groups that were cloned in the last clone cycle are destroyed (which you can then use to initiate new clone cycle)

Name	Value	Description
spawn? in? f? clone?	Name	Watches the flag <Name> (as accessed by DCS) for a change. **Each time the flag's value changes, a new clone cycle is initiated** You can use any of the synonyms for input (f?. in? spawn?, clone?), but only one per zone.
empty!	Name	When all units from the last clone cycle have been destroyed, this flag's value is changed according to method Note that this **includes all static objects** that are included in the template
deSpawn? deClone?	Name	Watches the flag <Name> for a change. Each time the flag's value changes, all units (including static objects) that are still alive from the previous spawn are removed from the game.

Note:

if you feed *empty!* directly into *spawn?* (set them both to the same value), you create an endless spawner that re-spawns the entire template after all previously spawned units are destroyed.

RESTRICTIONS

Currently, clone zones do not support units that you linked to ships in ME (e.g., a carrier with aircraft groups on them). This may change in later releases.

INTEGRATION WITH GROUPTRACKERS

CloneZones can automatically add any spawned groups (but not static objects) to a groupTracker. Simply use the "trackWith:" attribute and supply the name of the zone that contains the groupTracker. If the groupTracker is stacked on the same zone as the cloner, you can provide the wildcard "*" as zone name

CALLBACKS (API)
Cloners sport strong support for Lua-scripted callback. Prove a callback that matches the following profile

```
function clonerCallback(theZone, theReason, arguments)
```

and your code will be invoked whenever one of the following happens (as provided by theReason argument)

- `"will despawn group"`
  The cloner is about the despawn a group, usually as a response to a spawn with "preWipe" set to true.

  Note:
  At the start of a mission, all cloners that also serve as template despawn their originals, and will thus invoked this for each original group. WARNING: Your callback for the initial removal of originals is only invoked if you subsribe to callbacks before you run cloneZones.start().

  `arguments` contains the group that is about to be despawned

- `"will despawn static"`
  The cloner is about the despawn a static object, usually as a response to a spawn with "preWipe" set to true.

  Note:
  At the start of a mission, all cloners that also serve as template despawn their originals, and will thus invoked this for each original static. WARNING: Your callback for the initial removal of originals is only invoked if you subsribe to callbacks before you run cloneZones.start().

  `arguments` contains the static object that is about to be despawned

- `"did spawn group"`
  Whenever the cloner creates a group, this callback is invoked. Note that cloners can have more than one group per template, so this can be invoked multiple times per spawn cycle.
  `arguments` contains the group that was spawned.

- `"did spawn static"`
  Whenever the cloner creates a static, this callback is invoked. Note that cloners can have more than one static per template, so this can be invoked multiple times per spawn cycle.
  `arguments` contains the static that was spawned.

- `"spawned"`
  Whenever a spawn cycle completes, this is invoked.
  arguments.groups contains all swawned groups, arguments.statics contains all spawned statics. This invocation is provided separately to the per-group/static invocations so you can bulk-process the spawned items rather than one by one

- `"empty"`
  All units that were part of the previous spawn have been killed. Invoked at maximum once per spawn cycle. This is not invoked if preWipe is causing the units to be destroyed. `arguments` is empty

- `"wiped"`
  When a spawn cycle undergoes a pre-wipe, this is invoked. `arguments` is empty.

3.4.11.2 Dependencies
Clone Zones requires dcsCommon, cfxMX and cfxZones.

3.4.11.3 Module Configuration
This module does not need to be configured

3.4.11.4 ME Attributes

Name	Description
cloner	Marks this ME Zone as a clone zone. **Value of this attribute is ignored**, use it to describe this cloner's function. **MANDATORY**
source	The source for the clone template, must be the name of clone zone. When a clone cycle is initiated, the template is fetched from the source zone, and the units are then spawned around If this zone's center. If this attribute is present, this zone is not scanned for units to create a template from. If you supply **more than one template zone names**, separated by comma (e.g., "SAM 9 small, SAM 9 big"), each time a clone cycle is initiated, the clone zone picks one template by random. Defaults to <not present, zone scanned for units to create a template from>
turn	Degrees in which the clones are turned relative to the template's original position, relative to the zone's center. Defaults to 0 (zero)
moveRoute	If this attribute's value is true, all waypoints are move the same amount as the cloned units upon spawn. Only relevant if the zone is cloning another zone's template. When not present or false, all spawned units use the template's waypoints. Defaults to false
onStart	When set to false (default), the cloner will not spawn during start. Note that if this spawn zone is used to create a template, this results in an empty zone, as all units used for the template are destroyed during template creation. Defaults to false (**no spawn on start** of mission). **To spawn units at mission start, set this attribute to true.** **Deprecated**. Use a raiseFlag module instead

Name	Description
masterOwner	If not present, all cloned units retain the exact ownership of the units that the template was created from. If present, all cloned units are owned by the faction (red/blue/grey) that owns masterOwner. Note that using masterOwner, the coalition for the units that this cloner spawns can change during the mission. Defaults to <none, retain template ownership>
spawn? f? in? clone?	Flag to watch for a change. If the value of this flag changes, a new clone cycle is initiated Defaults to <none> This flag has multiple synonyms. Use only one per zone.
preWipe	If this attribute is true, any remaining units from the previous cloning cycle are removed from the game when the next clone cycle starts. Use this to 'refresh' groups like SAMs or Tanks that can run out of ammo. Default: false
~~empty+1~~	~~The value of this flag is increased by one when all units that were spawned during the last clone cycle are destroyed. This includes static objects. Defaults to <none>~~ **Deprecated**. Use empty! instead
empty!	The value of this flag is changed according to method when all units from the last spawn have been destroyed, including all static objects. **Note:** use this in favor of empty+1 **Note:** use either empty+1 or empty! but **not both**.
method	Describes how empty! is to be set during a pulse: • on – set the flag to 1 (one) • off – set the flag to 0 (zero) • inc – increases the flag's value by 1 (one) • dec – decreases the flag's value by 1 (one) • flip – set the flag's value to 1 if it's currently 0 (zero), or to 0 (zero) in every other case. Defaults to "inc"
deSpawn? deClone?	Flag to watch for a change. If the value of this flag changes, the remaining units / static objects from the previous spawn are removed. Note that if you trigger deSpawn?, empty+1 will **not** trigger subsequently. Defaults to <none>
trackWith:	List of groupTracker zones. All spawned groups are added to these groupTrackers. If you have stacked the tracker on the same zone as the cloner, you can use a single asterisk '*' as zone name. Supports a comma-separated list of trackers if you simultaneously want to pass the cloned groups to multiple trackers, e.g. "GroundTrack, HeloTrack" This is useful if your cloner clones more than one group, and your trackers use filtering.

Supports DML flags
Supports Watchflags

3.4.11.5 API

3.4.11.5.1 spawnWithCloner(theZone)

Initiates a clone cycle for theZone. theZone must be clone zone.

3.4.11.5.2 getCloneZoneByName(aName)

Returns the clone zone that is named aName. If no such clone zone exists, nil is returned.

3.4.11.5.3 cloneZones.addCallback(theCallback)

Adds theCallback to the list of callbacks that are invoked whenever a cloner event occurs. theCallback should conform to the profile

```
function clonerCallback(theZone, theReason, arguments)
```

with theZone being the zone the cloner is anchored to, theReason a string describing why the callback is invoked, and arguments some additional reason-specific data.

The following reasons are defined (details see above)

- `"will despawn group"`
- `"did spawn group"`
- `"will despawn static"`
- `"did spawn static"`
- `"spawned"`
- `"empty"`
- `"wiped"`

3.4.11.6 Using the module

Include the cloneZones source into a DOSCRIPT action at the start of the mission. Create Clone Zones by creating a Trigger Zone and adding attributes as described above.

3.4.12 cfxObjectSpawnZone

3.4.12.1 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. **Each time the flag value changes, a new set ob objects is spawned**
activate?	Number	Watches the flag <Number>. Each time the flag's value changes, the spawner's 'paused' setting is forced to 'false'. Used to 'activate' a paused spawner
pause?	Number	Watches the flag <Number>. Each time the flag's value changes, the spawner's 'paused' setting is forced to 'true. Used to 'pause' a spawner

This allows mission designers to spawn objects whenever a player unit enters a zone (e.g. cargo containers for helicopters).). Like unit spawners, object spawners support watchflags that allow the mission to pause and un-pause (activate) spawners by changing a flag. This allows other modules (e.g. rndFlags) to activate a paused spawner, and turn it off at will.

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.

3.4.12.2 Dependencies
Required
ObjectSpawnZones requires dcsCommon, cfxZones

Optional:
cfxCargoManager (for managing cargo events).

3.4.12.3 Module Configuration
ObjectSpawnZones does not require any configuration

3.4.12.4 ME Attributes

Name	Description
objectSpawner	Marks this ME Zone as a spawn zone. **Value of this attribute is ignored**, use it to describe what it spawns to make mission editor easier for you **MANDATORY**
f? spawn? spawnObjects?	An ME-compatible flag (e.g. 100) that this object spawner 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. You can use any synonym, but only one per Zone
pause?	Flag to observe. Each time the flag's value changes, the spawner's 'paused' setting is forced to 'true. Used to 'pause a spawner
activate?	Flag to observe. Each time the flag's value changes, the spawner's 'paused' setting is forced to 'false'. Used to 'activate' a paused spawne

Name	Description
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. **WARNING**: 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) **MANDATORY**
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 objects 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)
baseName	Used to create the names that uniquely identify the objects that are spawned to DCS. **If provided, MUST BE UNIQUE** 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 or query flag f?). 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

Supports DML Flags

3.4.12.5 API

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

3.4.12.5.1 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)`

3.4.12.5.2 getSpawnerForZone(aZone)

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

3.4.12.5.3 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.

3.4.12.5.4 getRequestableSpawnersInRange(aPoint, aRange, aSide)

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

3.4.12.5.5 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.

3.4.12.5.6 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.

3.4.12.6 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

3.4.13 cfxCargoReceiverZone

3.4.13.1 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 **(Deprecated – use method and f!)**
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 **(Deprecated – use method and f!)**
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 **(Deprecated – use method and f!)**
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 **(Deprecated – use method and f!)**
f! cargoReceived!	Number	Sets this flag according to method each time cargo is delivered into the zone

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`

3.4.13.2 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

3.4.13.3 Module Configuration
(none)

3.4.13.4 ME Attributes

Name	Description
cargoReceiver	Marks this zone as a cargo receiver zone. Value is ignored **MANDATORY**
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 **(Deprecated – use method and f!)**
f=0	Sets the flag specified in Value to 0 when cargo is delivered **(Deprecated – use method and f!)**
f+1	Increases the value of the flag specified in Value by 1 when cargo is delivered **(Deprecated – use method and f!)**
f-1	Decreases the value of the flag specified in Value by 1 when cargo is delivered **(Deprecated – use method and f!)**
method cargoMethod	Describes how flag! is to be set during a pulse: • on – set the flag to 1 (one) • off – set the flag to 0 (zero) • inc – increases the flag’s value by 1 (one) • dec – decreases the flag’s value by 1 (one) • flip – set the flag’s value to 1 if it’s currently 0 (zero), or to 0 (zero) in every other case. Defaults to “flip”
f! cargoReceived!	The flag to bang! when the object is destroyed. Use only one synonym per zone.

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

3.4.13.5.1 addCallback(cb)

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

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

3.4.13.6 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

3.4.14 cfxArtilleryZones

3.4.14.1 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.

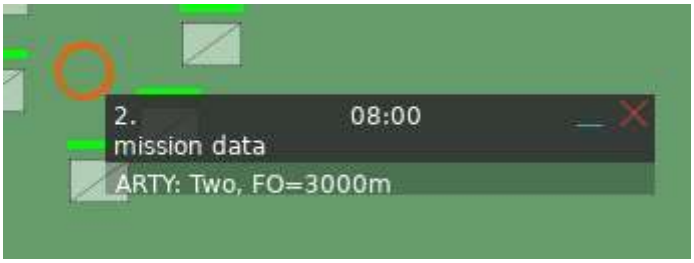
ME Flag Integration

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? in? artillery?	Number	Watches the flag <Number> (as accessed by DCS) for a change. **Each time the flag value changes, a new fire cycle is initiated**

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)
```

3.4.14.2 Dependencies
cfxArtilleryZones requires dcsCommon and cfxZones

3.4.14.3 Module Configuration
None.

3.4.14.4 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
artilleryTarget	Marks this zone as an artillery zone. Value is ignored **MANDATORY**
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 **each** 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 **true**.
shellVariance	Difference in shell's explosion power, in percent. Defaults to 0.2 (20%)
f? in? artillery?	**DML Watchflag**. When triggered, the artillery bombardment starts. Defaults to <none> You can use any synonym, but only one per zone
triggerMethod artyTriggerMethod	Defines the trigger condition for the DML Watchflag. Defaults to "change"
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

Supports DML Flags
Supports Watchflags

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

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

3.4.14.5.1 addCallback(theCallback)
Adds theCllback to the list of callbacks. Your callbacks will be invoked whenever a fire or impact event occurs.

3.4.14.5.2 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.

3.4.14.5.3 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

3.4.14.5.4 findArtilleryZoneNamed(aName)

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

3.4.14.5.5 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.

3.4.14.5.6 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

3.4.14.5.7 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.

3.4.14.6 Using the module

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

Place artillery zones with ME

3.4.15 cfxOwnedZones

3.4.15.1 Description



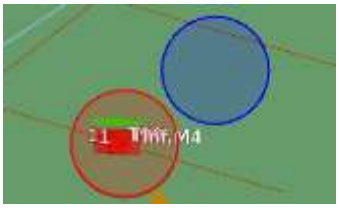
Owned Zones is a module that managers 'conquerable' zones that spawn attackers and defenders, and that keeps a record of which coalition owns which zone. Ownership is updated regularly. Owned Zones anchors itself to zones with an ‘owner’ attribute from ME.

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 *and including this module* 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.

Note that these bang! flags are **set in the zone’s configuration zone**, as they apply to all owned zones.

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. Applies to all owned zone, is set in the module’s config 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. Applies to all owned zone, is set in the module’s config 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. Applies to all owned zone, is set in the module’s config zone

Additionally, Owned Zones supports changing flags on **individual zone** basis.

Name	Value	Description
conq+1	Number	Increase this flag by one each time this zone is conquered. This flag applies to one zone (that one that carries this attribute)

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)

3.4.15.2 Dependencies
Required: dcsCommon, cfxZones

Optional: cfxGroundTroops

3.4.15.3 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

3.4.15.4 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) **MANDATORY**
conq+1	Increase this flag’s value by 1 (one) each time it is conquered. Defaults to <none>
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”

Name	Description
	places two Infantry soldiers. **Warning:** these types need to *exactly* 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. **Warning:** these types need to *exactly* 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. **Warning:** these types need to *exactly* 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. **Warning:** these types need to *exactly* match DCS's types. Be sure not to accidentally insert blanks. 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"

3.4.15.5 API

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

3.4.15.5.1 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)
```

3.4.15.5.2 getOwnerForZone(aZone)

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

3.4.15.5.3 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.

3.4.15.5.4 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

3.4.15.5.5 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

3.4.15.5.6 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

3.4.15.5.7 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

3.4.15.6 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.

3.4.16 FARP Zones

3.4.16.1 Description

FARPZones is a Zone Extension that improves in-game FARP capabilities. It automatically creates all “resource” 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 static object itself, so that the Resource Vehicles are easy to place, and you can ensure that the FARP is contained within the Zone.

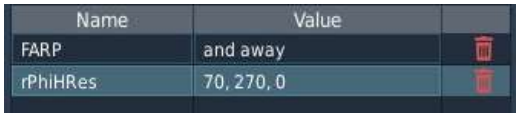
Warning
You should only place **at most one FARP Zone over a FARP**. Since the FARP Zone associates itself with the nearest FARP, make sure to place the FARP Zone near the FARP in question. If you place two FARP Zones close to the same FARP, there will be no error. When catured, though, the FARP can behave unpredictably, as it is not defined which FARP Zone will receive the notification.

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 or resource vehicles.

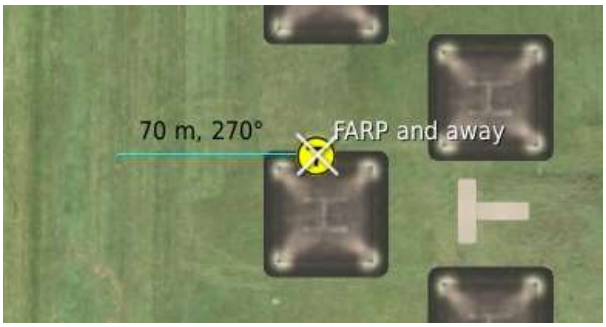
Note:
FARP Zones do not work with Airfields.

r-Phi-What???



In order to place the resource and defensive units, FARP uses a system called ‘polar notation’. It’s very easy to use in ME, and you’ll soon find out why. Let’s start with placing the resource vehicles: they deploy as a line around the resource point. On the left you can see that we have told this FARP that it should deploy the resource vehicles at “70, 270, 0”. This means the following: the resource vehicles are to deploy around a point that is 70 meters distant from the zone center (that’s “r”), at a bearing of 270 degrees (Phi, from the center), and that the units all deploy with heading (H) zero. Hence rPhiH = “70, 270,0”

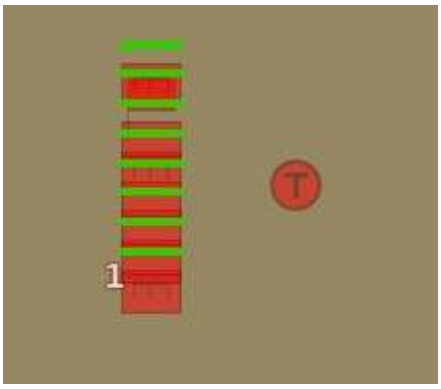
But how on earth did we arrive at these numbers, and why are we using such a strange way to describe a simple offset? Because we can get those numbers directly from ME, inside ME!

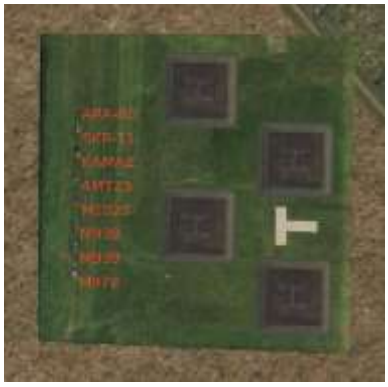


Using the ruler tool  in ME, we can, when measuring from the zone’s center, directly get both R (70 m) and Phi (270°) for

the location to place the units. Since the line of vehicles should be facing north, that's the final 0 (you can use the ruler a second time to measure the heading the line should have).

The result is this in-game:

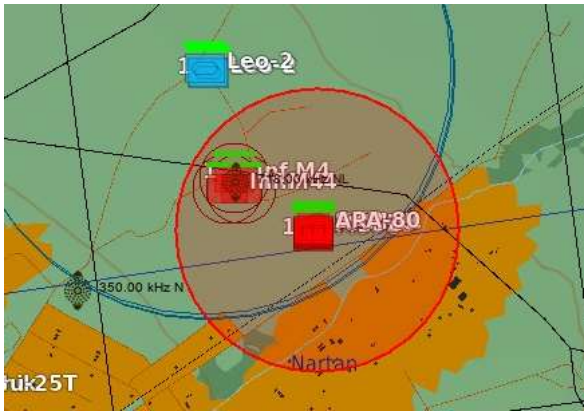




We follow the same procedure when placing the defenders that are spawned for the FARP when captured. The defenders always deploy inside a 100 meter radius, and you place the center of that circle with "rPhiHDef". If the FARP is captured, the defending vehicles for the new owning faction uses the same spot. Note that using ME's ruler to measure the spot for the zone you can achieve very precise positioning of defenders, utilizing the surrounding terrain.

F10 Map Display Options

FARP zones are displayed on the F10 map in-game, and are colored by their owning faction (red/blue/grey). You should place the FARP Zone very close to the actual FARP unit to make the circle correctly reflect the FARP's conquer zone, as the circle created on the map is exactly 2000m in radius and then reflects the area that when entered by enemy ground forces, captures the FARP.



Since it's not always desirable for FARPS to be visible, or only visible when owned by a specific side, each FARP has individual attributes that control when a FARP Zone is drawn on the F10 Map. Use "hideRed", "hideBlue", "hideGrey" for that purpose. They all default to 'false', resulting in a FARP being visible by default.

FARP Ownership and Spin-Up

FARP Ownership is governed by DCS, and FARP Zones always reflect that. When a FARP is conquered, FARP Zones detects this and starts the 'spinUp' cycle (a wait cycle). When that cycle is complete without another change in ownership, the FARP becomes operational for the conquering side: the resource and defense vehicles are spawned at the location that is indicated by the rPhiHRes attribute (see above and below)

When the mission starts up, no spin-up is required, all resource vehicles spawn immediately, the FARP can immediately to for all services.

FARP Services
For many mission designers, providing services at a FAPR (e.g. Repair, Rearm, Refuel) can be an issue. FARPZones always ensure that a FARP has all required service vehicles ready for the FARP to immediately (or after spin-up when captured) provide all services. As mission designer it suffices if you add a Zone and the FARP attribute, and the FARP has all services available. Note, however, that the resource vehicles are not automatically replaced by the FARP, the player must defend them or the services may nor be available any more.

Interaction with other DML modules
FARP Zones interact automatically with most other modules when required. Here are some hints when you plan to use other Zone Enhancements with FARP Zones

- SSB Client – integration is fully automatic. SSB Client recognizes which FARP belongs to what coalition and blocks slots accordingly. No conflict
- SpawnZones – since a FARP can be conquered, you should be careful when using spawn zoned inside a FARP zone. Make the spawner's masterOwner the FARP to prevent the spawner from spawning when the FARP is owned by another faction.

3.4.16.2 Dependencies
Required: dcsCommon, cfxZones

3.4.16.3 Module Configuration
To configure FARPZones via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is off
spinUpDelay	Number of seconds after a capture that the FARP becomes active (the resource vehicles spawn). Defaults to 30

3.4.16.4 ME Attributes

Name	Description
FARP	Indicates that this zone is a FARP zone. Value is ignored. **MANDATORY**
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

Name	Description
formation	Formation of the defenders group. See dcsCommon for supported group formations. Defaults to ‘circle_out’.
rFormation	Radius of the circle that the defenders assemble in. Defaults to 100m
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.

3.4.16.5 API
None.

3.4.16.6 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.

3.4.17 cfxMapMarkers

3.4.17.1 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).

3.4.17.2 Dependencies
Required: dcsCommon, cfxZones

3.4.17.3 Module Configuration
No special configuration required

3.4.17.4 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” **MANDATORY**
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”

3.4.17.5 API
None.

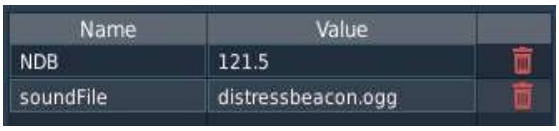
3.4.17.6 Using the module
To enable, add the script to the mission as a DOSCRIPT action during Mission Start

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.

3.4.18 cfxNDB

3.4.18.1 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.



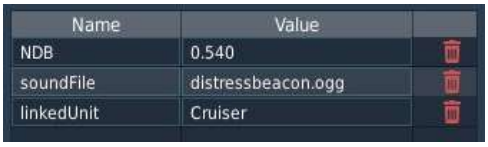
NDB are exceedingly easy to set up – all they need is a frequency and a sound file (since DCS currently does not support a set of default sound files, you must supply your own. The 'ADF and NDB fun" mission includes a small (public domain) sound file you can use to simulate an ELT signal.

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 unliked NDB.

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 l10n/DEFAULT/. If you manually place sound files in your mission at other places than ME's default location ("l10n/DEFAULT/"), you must provide the path to that location yourself, relative to l10n/DEFAULT/.

ME Integration
You can turn an NDB on and off at any time in the mission using normal ME flags. You tell the NDB zones which flags to watch, and every time that those flags change values, the NDB is turned on or off accordingly.

Name	Value	Description
on?	Number	Watches the flag <Number> for a change. **Each time the flag value changes, the NDB is started (will also cause the transmission sound to rewind).** The current paused value is ignored, and then set to false after the NDB has started
off?	Number	Watches the flag <Number> for a change. **Each time the flag value changes, the NDB is stopped.** paused is set to true for this NDB

3.4.18.2 Dependencies
NDB requires dcsCommon and cfxZones.

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

3.4.18.3 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 *for moving NDB* (i.e. an NDB with a linkedUnit attribute). **Note** 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 *from the beginning*. 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

3.4.18.4 ME Attributes

Name	Description
NDB	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. **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)** **MANDATORY**
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 (l10n/DEFAULT/). If you use ME to import the sound files, you do not have to specify the location. Remember to import the sound file into the mission else no sound will play.
watts	Transmission power (in watts) for the NDB. 100 Watts usually has a range of some 150 km. Defaults to 100 Watts
paused	If set to true, on mission start the NDB will not start up. Use the "on?" watch flag attribute or API to turn it on. Defaults to false
on?	Watchflag. **Each time the flag triggers, the NDB is started (will also cause the transmission sound to rewind).** The current paused value is ignored, and then set to false after the NDB has started. Defaults to no flag to watch
off?	Watchflag. **Each time the flag triggers, the NDB is stopped.** paused value is set to true after the NDB has stopped. Defaults to no flag to watch
triggerMethod ndbTriggerMethod	Defines the trigger condition for DML Watchflags. Use only one synonym per zone Defaults to "change"

3.4.18.5 API

You can use the API to easily start and stop an NDB from script. Note that you must create the NDB in-mission with ME first,

3.4.18.5.1 startNDB(theNDB)

Starts an NDB as defined by theNDB. Any paused attribute is ignored, and paused is then set to false. theNDB can be a string referring to the zone by name, or the NBD zone itself.

3.4.18.5.2 cfxNDB.stopNDB(theNDB)

Stops an NDB as defined by theNDB. The paused attribute is set to true. theNDB can be a string referring to the zone by name, or the NBD zone itself.

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

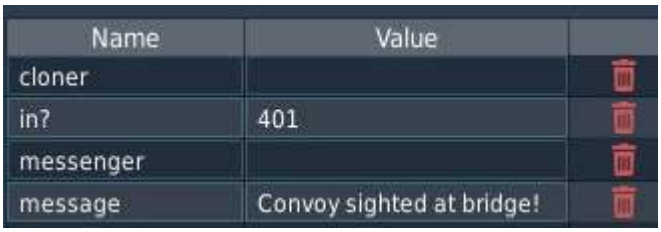
To use, simply add the ‘NDB’ and ‘soundFile’ attributes to a zone.

To prevent the NDB to start transmitting when the mission starts, add a ‘paused’ attribute and set it to false. You can then start the NDB using the on? watch flag.

To stop an NDB from transmitting, use the off? Watch flag.

3.4.19 Messenger

3.4.19.1 Description



Created purely for convenience, and expanded for perfection, the messenger module is an improved replacement for ME Triggers with a MESSAGE TO ALL or MESSAGE TO COALITION action. A big convenience factor is that you can visually package a messenger into a zone with other modules like cloners. That way you can easily identify what message belongs to which action. When stacking with other modules, make sure to use a shared input synonym for both (in the example to the right, messenger and spawner share the 'in?' attribute as input, so the message is displayed at the same moment that the cloner spawns.

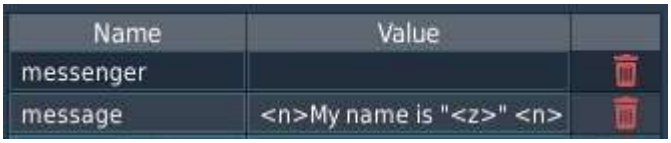


Messengers simply display a message and/or play an audio file to the entire world, or a coalition when triggered. Since DML triggers fire when a value changes, they are slightly easier to set up by themselves for repeated uses, and dramatically easier to use

Unlike MESSAGE TO XXX, a messenger module can be enabled and disabled by flags, making it very easy to create message controlling logic in your mission (for example simulating a unit that stops broadcasting text/audio messages when it is destroyed).

MESSAGE WILDCARDS

The message text supports a number of wildcards, that get replaced at run-time with their current values, allowing you some limited dynamic text:



- <n> creates a line feed
- <z> – this is replaced with the zone's name as given in ME
- <v> – this is replaced with the current value of the flag that is specified in the messageValue? Attribute



ME INTEGRATION

Name	Value	Description
messenger?	Name	Watchflag. When triggered, the module will display the message and/or play sound
~~f?~~ ~~in?~~ ~~messageOut?~~	~~Name~~	~~Watches the flag <Number> for a change.~~ **~~Each time the flag value changes,~~** ~~the message is displayed and sound is played~~ ~~You can use any synonym, but only one per zone~~ **Deprecated. Use messenger? instead**

Name	Value	Description
messageValue?	Name	The currently held value of this flag can be displayed in message by using the <v> wildcard
messageOn?	Name	When the value of this flag changes, the messenger is turned on. If it already was on, nothing happens. All messengers start in the 'On' state and require at least one signal on 'messageOff' to disable it.
messageOff?	Name	When the value of this flag changes, the messenger is turned off. Any further messages are suppressed. If the messenger already was turned off, nothing happens.

3.4.19.2 Dependencies
Messenger requires dcsCommon and cfxZones to run

3.4.19.3 Module Configuration
To configure the messenger module via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is false

3.4.19.4 ME Attributes

Name	Description
messenger? **messenger**	Watchflag. When triggered, the module will display the message and/or play sound. **MANDATORY**
~~f?~~ ~~in?~~ ~~messageOut?~~	~~When triggered, the message is displayed, and sound is played~~ ~~You can use any synonym, but only one per zone~~ **Deprecated. Use messenger? instead**
message	The text of the message to be displayed **TEXT WILDCARDS** <n> creates a new line <z> is replaced with zone's name <v> is replaced with the value currently held by the flag specified in *messageValue?*
triggerMethod msgTriggerMethod	Defines the trigger condition for DML Watchflags. Use only one synonym per zone Defaults to "change"
messageValue?	A flag name that is read whenever a message is put out. The value that this flag contains can be accessed with the <v> wildcard in the message attribute
clearScreen	If true, erase all existing messages. Defaults to false
~~spaceBefore~~	If true, an empty line is issued before the text message.

	Defaults to false **Deprecated. Use <n> in the message instead**
~~spaceAfter~~	If true, an empty line is issued after the text message. Default is false **Deprecated. Use <n> in the message instead**
soundFile	Name of the sound file (including extension like '.wav') that is to be played. Defaults to '<none>. Note that the sound file's name must be specified relative to the mission's default location for sound files (l10n/DEFAULT/). If you use ME to import the sound files, you do not have to specify the location. Remember to import the sound file into the mission else no sound will play.
coalition msgCoalition	The coalition that the message/sound is played to. If no coalition is given, text and sound are played to all. Legal values are "red", "blue", "neutral", 0, 1, 2 Default is <none> (play to all).
messageOn?	When the value of this flag changes, the messenger is turned on. If it already was on, nothing happens All messengers start in On state and require at least one signal on their messageOff input to disable. Defaults to <none>
messageOff?	When the value of this flag changes, the messenger is turned off. Any further messages are suppressed. If the messenger already was turned off, nothing happens. Defaults to <none>

Supports DML Flags

3.4.19.5 API
No API. Use trigger.action.outText() and trigger.action.outSound() instead

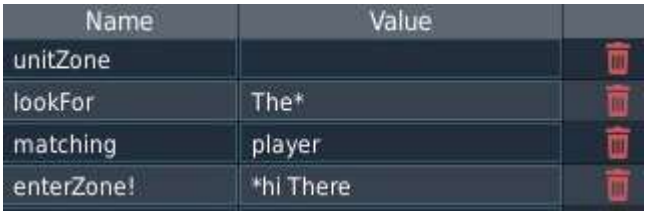
3.4.19.6 Using the module
To enable, add the script to the mission as a DOSCRIPT action during Mission Start

To use, simply add the 'messenger attribute to a zone.

3.4.20 Unit Zone

3.4.20.1 Description

This is the DML version of trigger zones – editable on the spot, with the ability to trigger DML and ED flags. Since you edit the trigger conditions right there on the map (well, inside the Trigger Zone's editor), it's much easier to directly connect the zone's purpose and function.



Unit Zones support wildcard matching of groups and player units: specify the string that a group's (or player unit's name) name starts with, and all groups that match that description automatically are checked. For example, with "Ta*****" you tell a unitZone to check all living groups whose name starts with "Ta" if they are inside the zone. If your mission contains the groups "Tank Busters-1102", "Tassel" or "Tamara's Revenge" they will all be checked.

This is particularly helpful to integrate zone testing for units that have been spawned with spawners and cloners, as these all use 'base names' when spawning groups – their group names all start with the base string, and you can check against that.

HOW UNIT ZONES WORK

For better integration and high performance, Unit Zones work slightly different from classing ME zones when they detect if a specific group is inside, so it helps to understand their detection logic.



- *Change based*
  Unit zones are designed to be used to detect a status change rather than a condition. This means that unitZones change a flag when the status of the entire zone changes. A unitZone has exactly two states:
  - **None** of the units to look for is inside the zone ("empty")
  - **Any** of the units to look for is inside the zone ("units present")

  Let's assume a unit is empty. The first unit of all the units that this unitZone is looking for that enters the zone creates a status change event (and flag change) for changing from "empty" to "units present". When subsequently other interesting units enter the zone while it is still in "units present" state, no state change is triggered, even if the unit that originally triggered the change leaves the zone.
  When the last interesting unit leaves the zone (or is killed inside the zone), the zone changes back from "units present" to "empty", and a flag change is initiated.

- *State is Initialized at Mission Start*
  UnitZones establish their initial state at the beginning of a mission. This initial state does not create a status change event, even if the initial state is "units present". Care must be taken when dealing with client (player) units: these are not present in the mission when it starts up, so if you design a mission with player units spawning

inside a unitZone, their appearance after mission start can trigger a unit zone.

- *Name-based matching with wildcards*
  UnitZones look for groups by name. Since one of DML's most powerful features is cloning, and cloned groups use a naming scheme based on the template's original name, unitZones support 'wildcard' name matching: you can either specify the full name of a group to look for (only a single match possible), or you can provide a base name and add an asterisk "*", telling unitZones that all groups whose name starts with the same string (minus asterisk) are to be regarded. This allows for powerful zone testing, even without using clones.

  WARNING
  When set to groups, unitZones matches **group names**, not individual unit names. When set to player (see below), it matches **player unit names**, not group names.

- *Discrete rather than continuous checks*
  unitZones tests for units present in a zone at regular time intervals (can be changed with a config zone). By default, this check is performed once per second. This has little consequences for ground units; as zones usually are large enough to make no difference. For fast moving units, however, there is the possibility that the cross the entire zone inside that time limit, so make sure that the zone is large enough. A plane that moves 500 mph crosses 740 feet in one second, so a 1000 feet diameter (500 ft radius) may barely be enough (500 km/h gives 140 m/s, requiring a 100m min radius). Since units rarely cross a zone in an optimal way, you may want to make the zone significantly larger (but you have encountered and resolved that same issue in classic ME before, so I won't re-tread that ground)
  Be advised that even though unitZone's unit matching is fast and low-impact, simultaneously adding many unitZones, lots of groups and indiscriminately jacking up testing (e.g. once every 10 ms) can degrade a mission's performance.

- *Red and Blue only*
  unitZones only works with red and blue groups and completely ignores neutral groups. If you need neutral groups to trigger an event, you'll have to resort to classic ME

- *Filtering*
  Usually, your unitZones are looking for highly specific units, and you know which category and/or coalition they belong to. UnitZones allows you to limit unit matching to categories (e.g. aircraft) and/or coalition. Use this to remove the risk of false positives (we all use copy/paste), and slightly increase performance.
  Note that unitZones coalition filtering may work slightly different from what you might expect: unitZone never inspects neutral groups, only red or blue forces. So, when you supply '0' or 'neutral' as filter, this will cause unitZone interprets this as 'both' and inspects red and blue (the same as if the attribute wasn't given at all).

- *Player vs Groups*
  Since all missions revolve around players, unitZones provide a special switch to only regard player units. Care must be taken to remember how unitZones match names. In the (default) *group* matching, all units whose group name matches the lookFor

attribute are checked. When set to player, only player units are examined, and the lookFor match is performed for the unit's (rather than group's) name.

OUTPUTS
unitZones supports multiple output flags that it can change when the relevant conditions are met:

- *enterZone!*
  when the zone previously was empty, **the first unit** that matches this zone's match criteria that enters this zone triggers a change on this flag

- *exitZone!*
  when a zone previously had units present, this flag is changed when a test determines that the zone is empty. This can happen when the last interesting unit has departed or was killed.

- *changeZone!*
  Every time a change happens (one of the above), this flag is also changed.

A special, optional output "uzDirect" can carry the current state of the unitZone: 0 = none of the units we are interested in are inside the zone, 1 = at least one of the units is in the zone.

3.4.20.2 Dependencies
unitZone requires dcsCommon and cfxZones to run

3.4.20.3 Module Configuration
To configure the unitZone module via a configuration zone,

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

Name	**Description**
verbose	Show debugging information. Default is false
ups	Number of zone checks performed per second. 0.1 means once every 10 seconds, 2 means twice a second. Defaults to 1

3.4.20.4 ME Attributes

Name	**Description**
unitZone	Marks this ME Zone as an anchor for unitZone **MANDATORY**
lookFor	Name for the group or (player) unit to check zone status. If the last character in the name is an asterisk "*", exact matches and all group/unit names that start with that string (minus asterisk) are

Name	Description
	accepted, e.g. if you supply ”Hel*” all of the following would be accepted: • Hel • Hello World • Helo Rescue-1 • Hellfire You can use this feature to your advantage in conjunction with cloners or spawners, as these all produce groups with a known base name. If you only supplied “Hel”,only (without the asterisk “*”) only the group whose name exactly matches “Hel” is checked.
matching	What type of units to match. Currently supported are • group (default): look for group names • player – look only at player units and match their unit’s (not group’s) name against lookFor Default: group
coalition	The coalition (red/blue) units to check. Accepts 0, 1, 2, red, blue, neutral. Note that **0 (zero) and ‘neutral’ means ‘both’** in this case, neutral groups/units are never checked against the zone Defaults to 0 (both)
filterFor	Which unit categories to look for. If no attribute is given, **all** categories are checked against the zone (when their name pattern matches). When you supply a filterFor attribute, only that category is considered. Currently supported are • 0 (zero) or “aircraft” or “air” • 1 or “helo” or “heli” or “helicopter” • 2 or “ground” • 3 or “ship” • 4 or “train” Defaults to no filtering
enterZone!	Change this flag when the first unit (player) or part of all groups that match the criteria enters the zone
exitZone!	Change this flag when the last unit (player) of all groups that match the criteria have exited the zone (being destroyed counts as leaving)
changeZone!	Changes this flag whenever enterZone! or exitZone! are triggered
method uzMethod	DML Flag method for output. Use only one synonym per zone
uzOff?	Watchflag. When triggered, this zone will no longer perform checks. When already off, nothing happens
uzOn?	Watchflag. When triggered, this zone will resume checks. When already on, nothing happens
triggerMethod uzTriggerMethod	Method that determines when the watchflags should trigger. Default is “change”
uzDirect	When present, this flag (or flags) is always set to the current state of the unit zone: • 1 when one or more units in the zone • 0 when none of the indicated units in the zone. Default is <none>

Supports DML Flags
Supports Watchflags
Supports zone-local verbose

3.4.20.5 API

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

To use, simply add the ‘unitZone’ and required other attributes to a zone.

3.4.21 Group Tracker

3.4.21.1 Description

GroupTracker is a DML module that allows you to generate DML-style events whenever a group or set of groups is destroyed. If you have some ME experience, lookt at it as DML's version of the GROUP ALIVE and GROUP DEAD trigger conditions, with a couple of counters thrown in for good measure.

GroupTracker is a quite versatile, has miniscule performance impact, and can make mission design delightful – especially in conjunction with other modules like cloners, xFlag and messenger.

HOW GROUP TRACKER WOKS

Simply put, a group tracker works like this:

- It's always named after the zone it is attached to
- You hand one or more groups to the tracker. The tracker can filter and discard groups haded to it by categories if you want it to.
- GroupTracker regularly (e.g. once per second) checks all groups that it's told to track
- When a group was destroyed since the last time it checked, it can increment the flag "removeGroup!"
- When you add a new group to a tracker, it can increment the flag "addGroup!"
- You can output the current number of groups to the flag "numGroups!"

And that's it. Group Tracker does not care what kind of groups it trackes: air, sea, ground, nor which coalition they belong to. All it does its count the number of groups, and change flags when it detects something new or destroyed.

SO WHO NEEDS THAT?

Who would need such a basic module? Everyone. Like many simple things you only notice their absence once they are gone. While most of groupTracker's abilities can be achieved with other means, using groupTracker makes it much easier. Implicit in above's description are a couple of important features that you can use for your own mission:

Remember that in DCS, flags not just indicate true/false, but they can carry numerical information (e.g. the number 42). DML modules usually trigger when an input flag changes in value, and you usually set up modules to 'increment' a flag when a change occurs.

Looking at the output flags that a groupTracker module supports, and supposing that no other module is modifying those flags, groupTrackers flags also double as follows:

- addGroup!
  contains the running total of all groups added to the tracker (this can be much higher than the number of groups currently tracked)



- removeGroup
  contains the running total of the number of groups that were tracked by this tracker and have been destroyed up to now

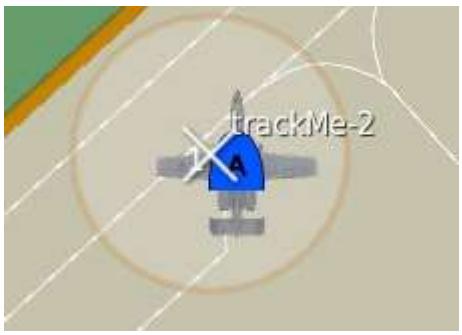
In effect, groupTracker not only allows you to be notified when a group is added to the tracker or a tracked group is destroyed, it also always gives you access to totals, and these are most often used to control scenario/victory condition or mission states. By providing the ability to count from a set, groupTrackers make short work out of otherwise difficult mission propositions, such as

- Provide a kill count
- Trigger reinforcements if 3 groups from a set of groups have been destroyed
- Disable spawners/cloners when a maximum number of groups is reached and re-enable them when it falls below a minimum
- Trigger if all tracked groups are destroyed

GETTING GROUPS ADDED TO A TRACKER
There are a number of ways to add a group to a groupTracker

- Place at least one unit of the group inside a zone with an "addToTracker:" attribute. Such units must exist at start of the mission, and are added to the listed trackers at mission start. The tracker that is referenced (in the example "plane tracker") must exist, or groupTracker will complain at start-up.





  Note I:
  This attribute can be added to any zone, and is only active when groupTracker is loaded. This attribute is only evaluated at mission start, so any group that does not have at least one unit inside this zone at start will not be added to the tracker (e.g. player units do not exist at mission start).

  Note II:
  Multiple groups inside the same zone are correctly processed

  Note III:
  addToTracker supports lists of trackers. Their names must be comma-spearated (","), so ensure that your zone names for trackers do not contain a comma.

- Some modules (those that spawn groups like spawners and cloners) support a "trackWith:" attribute. These groups are added dynamically, after they are spawned into the mission, to a tracker.

- Lua scripts can add groups by invoking the API at any time.

SPECIAL USES
Note that a groupTracker can act as a more capable version of a clone zone's empty! flag, and many missions use a zone stack that contains both a cloner and a groupTracker to supplement a spawner's inbuilt empty! ability to only trigger when *all* cloned groups (not merely the last batch) have been destroyed.

WILDCARD NAME SUPPORT
Since groupTrackers are often part of a stack and only track groups that are 'produced' by that stack, the "addToTracker:" and "trackWith:" attributes support the asterisk "*" wildcard: when you want to reference the groupTracker attached to the same zone, simply use an asterisk instead of the zone name. This allows for easy copy/paste of stacks.

FILTERING
groupTrackers support filtering when they are handed groups. By default, they accept all groups. Occasionally, you want a tracker to track only a certain category of units (e.g., ground troops), and you can turn this feature on with the tracker's 'groupFilter' attribute.

A common use for this is when a cloner spawns multiple categories, and you want to separate and track them separately.

LIMITATIONS
groupTracker does not track static objects nor map/scenery objects, and such objects can't be added with an "addToTracker:" or "trackWith:" attribute.

ME INTEGRATION
GroupTrackers main purpose is to create flag events (changes), and therefore provides heavy ME / DML flag integration:

Name	Value	Description
addGroup!	Name	Whenever a group is added to the tracker, the value of this flag is increased. If not changed by other modules, this flag also doubles as a running total of all groups added to the tracker
removeGroup!	Name	Whenever a tracked group is destroyed, the value of this flag is increased. If not changed by other modules, this flag also doubles as a running total of all watched groups that have been destroyed while they were tracked
numGroups!	Name	The value of this flag always represents the number of groups currently watched by this tracker. It's value is updated 1/ups times per second.

3.4.21.2 Dependencies
groupTracker requires dcsCommon and cfxZones

3.4.21.3 Module Configuration
To configure the groupTracker module via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is false
ups	Number of group checks performed per second. 0.1 means once every 10 seconds, 2 means twice a second. Defaults to 1

3.4.21.4 ME Attributes
Attribute to add all groups that are (even partially) covered by this zone to a groupTracker

Name	Description
addToTracker:	List of groupTracker zones. All groups that have at least one unit inside this zone are added to these groupTrackers. This happens only at mission start-up, and therefore only work for non-player-controlled planes (since player-controlled planes do not exist at mission start-up). If your player group contains AI planes, place one of those into the zone, and that group can be added to a tracker. If you have stacked the tracker on the same zone, you can use a single asterisk ‘*’ as zone name. Supports a comma-separated list of trackers if you simultaneously want to pass the groups to multiple trackers, e.g. “GroundTrack, HeloTrack” This is useful if the zone contains more than one group, and your trackers use filtering Add all groups that have at least one unit in this zone to the tracker whose zone name is given in the Value field.

Attributes for a groupTracker

Name	Description
tracker	Marks this zone as a groupTracker. It can be referenced by the zone’s name passed in the trackWith: and addToTracker: attributes. When referenced locally, a single asterisk “*” can be used as wildcard name for easy copy/paste of the entire stack **MANDATORY**
addGroup!	Whenever a group is added to the tracker, the value of this flag is increased. If not changed by other modules, this flag also doubles as a running total of all groups added to the tracker Defaults to <none>
removeGroup!	Whenever a tracked group is destroyed, the value of this flag is increased. If not changed by other modules, this flag also doubles as a running total of all watched groups that have been destroyed while they were tracked

Name	Description
	Defaults to <none>
numGroups!	The value of this flag always represents the number of groups currently watched by this tracker. **This value is updated 1/ups times per second**. Defaults to <none>
groupFilter	Which unit categories to track. If no attribute is given, **all** categories are tracked. When you supply a groupFilter attribute, only that category is accepted when attempting to add to a tracker. Currently supported are • 0 (zero) or “aircraft” or “air” • 1 or “helo” or “heli” or “helicopter” • 2 or “ground” • 3 or “ship” • 4 or “train” Defaults to no filtering

Supports DML Flags

3.4.21.5 API
Tbc

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

To use, first create a groupTracker by adding the ‘tracker’ attribute to a zone

Then add groups to trackers, use zones with an addToTracker: attribute, or cloners and spawners with a “trackWith:” attribute.

3.4.22 Wiper (Unit/Object removal)

3.4.22.1 Description

Wiper (with a 'w') is a module that can, upon activation remove specific objects that are inside the zone. DCS differentiates objects by categories, and wiper functions along these same lines. Currently, the following object categories are accessible through ME:

- `Unit` (1) – controlled units (ground, fixed-wing, rotor-wing, ships, trains), AI and player-controlled.
- `Weapon` (2) – usually munitions, usually only short life span.
- `Static` (3) – uncontrolled models that can look exactly like AI-controlled units, plus further objects that can be used to populate a map. DCS may by itself decide to convert an Unit to a Static object, just to cause grief to mission designers, see below.
- `Base` (4) – Most likely FARPs and landable ships. Unambiguous definition hard to come by
- `Scenery` (5) – some (but by far not all) of the carious objects that populate the map when it comes out of the box. And no. Trees are (sadly) not part of that definition. As aren't other, highly interesting objects that we'll touch upon later.
- `Cargo` (6) – hopefully the cargo that you are looking for.

There are points in your mission where you want to clear out objects, be it at the start of the mission, or later, for effect reasons. DML provides the wiper module for this purpose

ABILITIES

Wiper can remove all objects of a certain category (see above) that are within the zone. Before removing the objects, it can apply a name filter, so only objects that match the category and name restriction are filtered.

For analysis purposes, you can also set a wiper to report all objects it finds inside it's zone, by category when it fires.

SPECIAL USE CASE: Cloners & Spawners

A peculiarity with DCS is that in some cases, with some versions of DCS, the engine removes a (dead or almost-dead) Unit, and replaces it (probably for performance reasons) with a matching Static Object. This can wreak havoc in more ways than one, as it will no longer be removed when you dispose of the group (for DCS internal reasons, too arcane and boring to elaborate here). In situations where your mission heavily relies on spawners and or cloners this can lead to an over-abundance of burnt-out husk and wrecks, that did not get removed by a pre-wipe sweep from the cloners. This is where you can use wiper in addition to the preWipe attribute. For added safety, key the wiper to the cloner's group base name, and only those statics will be removed that were formerly AI Units spawned by the cloner.

LIMITATIONS

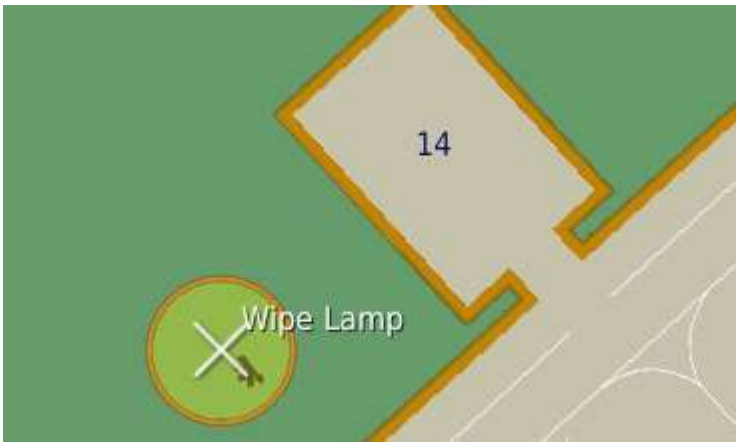
There are many intriguing peculiarities in DCS, and the way it treats the destruction cycle of units is among them. Another one are scenery objects:

- When units fight, some may get destroyed. Depending on how overwhelming their defeat and how strong they are, the transition from living AI unit to dead husk may be different. Infantry getting killed usually transitions from alive to dead by playing a short animation cycle, and then replacing the live Unit with a static object. Wiper can remove these without problems

- Stronger units may start burning, and then, after "cooking off" may explode or simply die, leaving different kinds of debris, flames, and smoke. Neither smoke nor flames can be removed by Wiper. Also, Wiper can only remove some of the debris left behind, while some wreckage may be left behind for the simple reason that it can not be detected. Also, in current DCS versions, the flames and smoke that can erupt from vehicles are not part of the 'detectable' world, and therefore not be removed by wiper modules. They will have to run their course until ED gives access to these effect objects in a later release

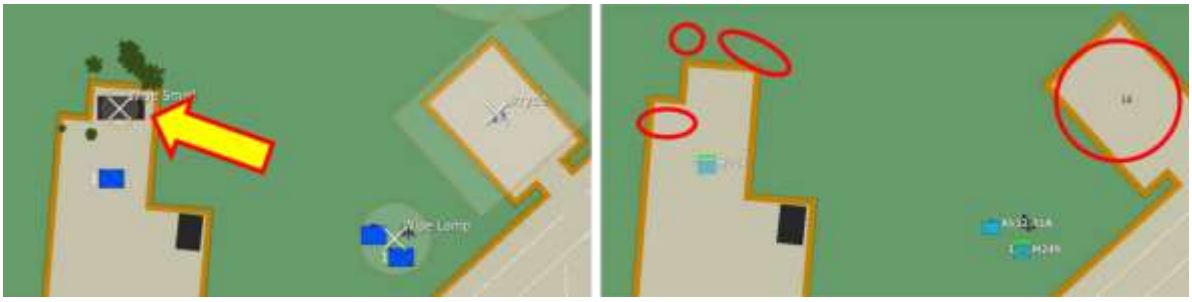


- There are scenery objects that are also invisible to wiper, meaning that they will not be removed when you put down a zone and attach the wiper module.
    - First to mention are trees: wiper can't see any trees, and therefore is currently unable to clear trees for you. Use the special zone ME ability to do that.
    - Then there are objects that, even though they may get an object ID when you right-click and choose 'Assign as' in ME; they do not appear to be visible to wiper (i.e. via world.seach()). For example, the lamp post close to parking lot 14 in Senaki Kolkhi. Place a wiper over the lamp, activate the inventory function, and marvel at the empty list – allegedly there is nothing there.



    - As mentioned above, most effects (fire, smoke) and debris will also not be visible to wiper, and hence can't be removed.

WARNING
When removing scenery objects, DCS's world.search() routine (that DML uses to query the world for objects to remove) can be exceedingly imprecise, and remove objects far outside of the wipe radius:



As you can see in the left image above, a small wipe zone is placed over the black rectangle that represents a building in Senaki Kolkhi (marked discretely by a yellow arrow with red outline). The wipe zone's radius only encompasses the building's center, nothing else.

In-game, however (right image), not only the building, but some surrounding trees, and – more disturbingly – the hangar for lot 14 have also been removed. This is not a DML bug, but a result of how DCS's internal methods work.

As a result, whenever you intend to remove some scenery or when you get unexpected results, run a wiper with the inventory option set to determine

- Which category the object you want to remove belong to, and more importantly,
- if the objects that you want to remove are even visible.
- When they are visible, selectively remove the objects with names (if possible).

ME INTEGRATION

Wiper's main use is to selectively remove objects inside the zone at the start, and other key moments in your game. To trigger object removal, simply use a flag.

Name	Value	Description
wipe?	Name	Watchflag that tells wiper when to trigger a remove cycle.
wipeInventory	yes/no	Tell the wiper to list all objects it found in the zone, by category prior to starting the wipe cycle.

3.4.22.2 Dependencies
Wiper requires dcsCommon and cfxZones to run

3.4.22.3 Module Configuration
To configure wiper via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is false
ups	Number of checks for a wipe trigger per second. Defaults to 1

3.4.22.4 ME Attributes

Name	Description
wipe?	Watchflag. Triggers a wipe cycle **MANDATORY**
triggerMethod triggerWiperMethod	Method that triggers the Watchflag Defaults to 'change'
category wiperCat	Category of the objects that are affected by the wipe (i.e. if they belong to the category they may be wiped). Possible values are • "unit" or 1 • "weapon" or 2 • "static" or 3 • "base" or 4 • "scenery" or 5 • "cargo" or 6 Defaults to 'static'
wipeNamed	*Optional* comma-separated name list that an object's name must match in order to be wiped. Supports an asterisk ("*") as wildcard to match anything. For example, "Ba*" would match "Base", "Ba", "Babushka", and "Bathyscape" **Examples:** • "Ba*" – all objects inside the zone whose name starts with "Ba" • "Grou*, Commander Kirk, He*" – all objects whose names start with "Grou" or "He", and the object whose name exactly matches "Commander Kirk" Defaults to <option off>, no name filtering
wipeInventory	A Boolean that turns on the wiper's inventory function. Whenever triggered, the zone lists all objects, it finds inside the zone, sorted by category. Note that there may be objects inside a zone that wiper cannot find, and that it may return objects that are not really inside the zone. Both are a DCS limitation, not a bug in wiper.

Supports Watchflags

3.4.22.5 API
(tbc)

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

To use, simply add the 'wipe? attributes to a zone and add other attributes to tell which objects inside the zone should be removed.

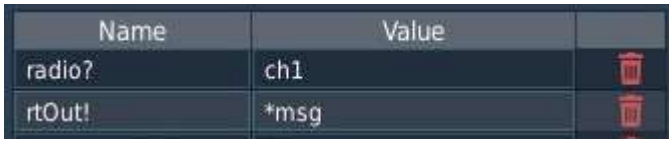
Then, to remove objects inside the zone, trigger the Watchflag. Note that you can trigger the wiper multiply times

3.4.23 Radio Trigger

3.4.23.1 Description

This small module provides a convenient, multi-use DCS-DML interface for ME's "Radio Item Add" feature. In ME, radio items can only set a flag to a certain value.





DML, on the other hand, triggers on changes. So standard ME radio items are inherently single-use (at least from the perspective of DML): once you have chosen a radio item, that flag is set to the new value (by ME), and can't be easily used again until it is re-set.

The DML RadioTrigger module simply converts setting a flag to a pulse that other DML modules can read, and **then resets the radio item flag** so that the next time the item is chosen from the communications menu, it will again trigger a pulse.

3.4.23.2 Dependencies

Radio Trigger requires dcsCommon and cfxZones

3.4.23.3 Module Configuration

To configure wiper via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is false
ups	Number of checks for a radio trigger per second. Defaults to 1

3.4.23.4 ME Attributes

Name	Description
radio?	Watchflag. Triggers a radio cycle, then resets this flag to its pervious value **MANDATORY**
triggerMethod radioTriggerMethod	Method that triggers the Watchflag Defaults to 'change'
method rtMethod	Method how the output flag should be triggered. Defaults to 'inc'
out! rtOut!	DML Flag to set when the module triggers Defaults to <none>

Supports DML Flags
Supports Watchflags
Supports zone-local verbose

3.4.23.5 API
(tbc later)

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

To use

- add the ‘radio? attribute to a zone and add other attributes to tell which flag to change how when the radio item is chosen
- In ME, add a radio item, and make it change the same flag as specified in the radio? attribute above.
- Connect any modules that need triggering by radio to the rtOut! attribute

3.4.24 baseCaptured

3.4.24.1 Description

baseCaptured is a module that generates output for DML flags when the nearest base (Airfield, FARP or Ships with airfield facilities) is captured (i.e., changes faction). Note that although possible in theory, currently DML does not support capturing ships. If it does, this function will also work for ships.

NEAREST BASE

When you add the baseCaptured! attribute to a zone, it automatically looks for, and then associates that zone with the nearest base (airfield, FARP/Helipad/Ship) it can find. As such, it is best to place that zone as close as possible to it's intended site, but DML will not mind if the airfield is hundreds of miles away. It will pick the closest one automatically. If you don't place the zone close the intended base, it is your onus to ensure that there are no closer bases, or the zone will associate with a different base.

Note that you can (and often find it helpful to) create multiple baseCaptured zones for the same base (one per faction).

OUTPUT

baseCaptured provides two very different outputs:

- Creates a DML Flag change when the nearest base is captured. These are the baseCaptured! redCaptured! and blueCaptured! signals. They provide DML Method-based signals when the base is captured. This is frequently used to trigger actions when a change of hands for this base happens.

- baseOwner continually provides a number (0, 1, 2) for the base's currently owning faction (0 = neutral, 1 = red, 2 = blue). This is frequently used to count the number one side holds and generate actions when that number falls below, or rises above a threshold.

3.4.24.2 Dependencies

baseCaptured requires dcsCommon and cfxZones

3.4.24.3 Module Configuration

To configure the baseCaptured module via a configuration zone,

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

Name	Description
verbose	Show debugging information. Default is false

3.4.24.4 ME Attributes

baseCaptured!	Marks this zone as a baseCaptured zone. It lists the flags that should be banged! when the closest base (FARP, Airfield, Ship with Helipad) to this zone is captured by another faction. **MANDATORY**
method captureMethod	DML method for output flags Defaults to 'inc'
blueCpatured! blue!	Flags to bang! when blue faction captures the closest base to this zone Defaults to <none>
redCaptured! red!	Flags to bang! when blue faction captures the closest base to this zone Defaults to <none>
baseOwner	Flag that is set by the module to the current faction (0 = neutral, 1 = red, 2 = blue) that currently holds this base.

Supports DML Flags
Supports zone-local verbosity

3.4.24.5 API
Tbc

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

To use

- add the 'baseCaptured!' attribute to a zone. This zone is automatically be associated with the nearest base.
- Add other attributes as needed

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**
  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**
  Generates civillian (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 (doc coming…)
  A slot blocking mechanism that blocks aircraft after they have crashed.

- unGrief (doc coming…)
  A griefer-repellant. Allow a number of friendly kills before this module will retaliate after each further kill. Intended for servers who suffer from abuse by immature users.

- **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 unit types (e.g. a BTR-80 kill yields 20 points), named groups (all units in the group named “special forces” yield 25 points) or named individual 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 achieved 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.

Tabulating 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, the group’s name that it belongs to, or its type) yields that score, and negative twice that amount if it was a fratricide. Name score has precedence over group score 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, group 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

(**Note:** static objects and scenery objects do not return a default score. If they are not listed in the score table, killing them does not award points).

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

Name	Description
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:

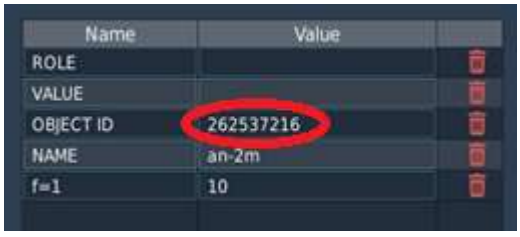
The score table uses the following format:

Name	Description
<tape or name> Type Exampe: BTR-80 Name Example: Big Kahuna	<Score as number> Example: 15 Example: 130 Note: The name is first checked again a unit's name, and then against the unit's group.

Static Objects:
To have static objects return a score, use the name given at the top of the static's info panel (where usually the group's name is assigned for AI units).



Scenery Objects
To assign score to scenery objects, use the OBJECT ID as name in the score table. The object id is shown when you right-click on a scenery object and then click on the 'assign as…' pop-up menu. Note that the object id can change between



DCS versions (when the map is updated), so use scenery score sparingly (use static objects instead).

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 in 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 (tbc)

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.

The module supports directing players to their CSAR mission targets and ‘live updates’ 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”.

ME INTEGRATION

You place CSAR Zones on the map that generate CSAR missions when the main mission itself starts up, or when signaled with a flag. Unless a ‘deferred = true’ attribute is present in a CSAR Zone, it generates a CSAR mission on startup. The CSAR Zone also generates a new CSAR mission every time the value changes on the startCSAR? flag

Name	Value	Description
in? startCSAR?	Name	Watches the flag <Name> for a change. **Each time the flag value changes, a new CSAR Mission is created.** Defaults to <none>

API Integration

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.

3.5.4.2 Dependencies
Required: dcsCommon, cfxZones, cfxPlayer, nameStats, cargoSuper

Optional: limitedAirframes

3.5.4.3 Module Configuration
To configure the csarManager module via a configuration zone,

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

Name	Description
verbose	Set to true to turn on debugging information. Defaults to false
ups	Updates per second. Defaults to 1
useSmoke	When approaching a mission target, activate smoke or not. Smoke can have significant performance impact when close in a helicopter, so the
smokeColor	Color of the smoke to pop when helicopter is in range and enabled. Understands numbers (0-4) and names (green, blue, etc)
csarRedDelivered!	Flag to change when a red coalition-flown CSAR mission ends successfully
csarBlueDelivered!	Flag to change when a blue coalition-fown CSAR mission ends successfully
csarDelivered!	Flag to change when a CSAR mission ends successfully
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 frequeny. 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
vectoring	If set to false, the mission report will no longer give range and bearing to the downed pilots. Default is true (range and bearing is provided)

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
CSARBASE	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. **MANDATORY**
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 and when signaled via startCSAR? flag, a new active CSAR missions is created.

Name	Description
CSAR	Identifies this as CSAR Zone that is converted into a CSAR mission upon mission start or when the value of the startCSAR? flag changes. **MANDATORY**
coalition	Faction (red/blue) for which this mission is generated
name csarName pilotName	Name of this mission, recommended is to use a personal name, e.g. "Lt. Wesley Crasher"

victimName	
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)
deferred	If true, CSAR missions are only created when the startCSAR? flag changes. Default is false (a CSAR mission is automatically created when the main mission starts up)
in? startCSAR?	When the value of this flag changes, a new CSAR is created according to this Zone's attributes. Defaults to <none>

Supports DML Flags

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. These can spawn CSAR missions on start and later using flags
- To create CSAR missions while the mission is running, see the API section or use the startCSAR? flag defined per CSAR Zone

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):

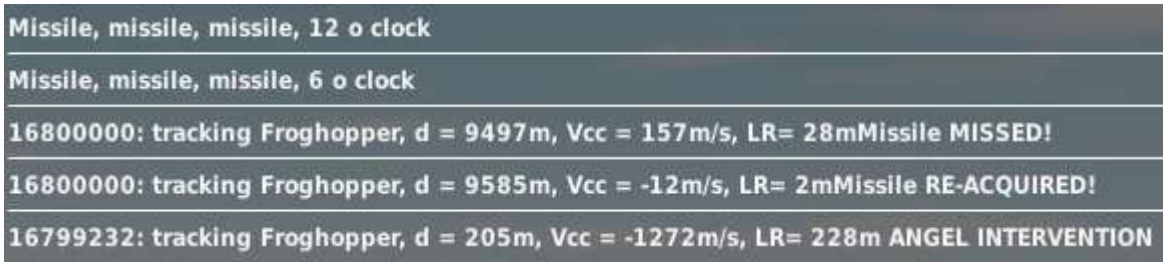
- Automatically protects player aircraft (fixed-wing and rotor-wing)
- Warns of missile launch with direction
- Destroys missiles shortly before they hit a protected unit
- Can announce warnings and 'interventions' to all or privately to the unit only
- Can protect player and AI planes
- Can add visual 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).

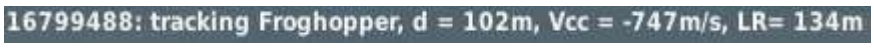
WARNINGS

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



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:



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 even 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 cannot 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
	WARNING I Even though this explosion is usually harmless for the protected plane, it can pose lethal to any other plane (wingmen). **WARNING II** 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. If you set this value to see explosions, make the value 1.0 Defaults to -1 (off)
fxDistance	When using explosions, this is the distance (in meters) away from the aircraft where the (real) explosion is going to take place. Defaults to 500

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

Civ Air is a high-performance drop-in module to quickly add civilian (or neutral) AI flights to missions. You control which airfields have civilian traffic, how many planes ferry between the airfields, and which planes to as civilian (note that DCS currently only has few truly civilian planes). If you simply add the module to your mission, and do not change anything, you will have up to ten (10) civilian flights between any two airfields on the map. The map and airfields are detected automatically.



HOW IT WORKS

Civ Air works by generating civilian (neutral, non-aggressive) flights between airfields. Once a flight ends (either by reaching the destination and parking, or crashing), a new flight is generated. This continues until the mission ends.

To generate a flight, Civ Air goes through a number a steps:

- Pick an aircraft type (e.g. Yak-40). It will always be part of a neutral faction.
- Pick a starting airfield
- Pick a destination airfield
- When the mission starts up, half the maximum number of planes start in the air above their departure airport. They all spawn immediately
- All other civilian planes spawn on the tarmac and perform a cold-start. Only one civilian flight spawns per cycle (once every 25 seconds unless changed with a config zone).

AIRCRAFT TYPES

Civ Air by default uses a set of air frames (identified by their type) and randomly picks one to create the flight. You can completely customize which frames are available (and add other – even non-civilian – airframe types) with the config zone. By default, Civ Air picks a type from the following list:

- Yak-40
- C-130

- C-17A
- IL-76MD
- An-30M
- An-26B

When you customize the list, you can increase the likelihood of a plane appearing by adding its type multiple times, increasing its chance of being picked over the others.

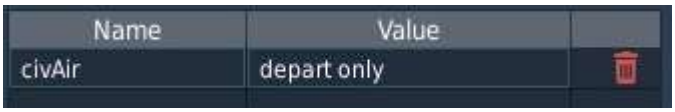
DEPARTURE/DESTINATION AIRFIELDS IN CIVAIR

Flights always originate from one airfield and end at another airfield (that is to say, they always remain on the map. CivAir currently does not support off-map departure or destinations; this may be added at a later time). The airfields that civilian flights depart from or fly to are taken from sets of airfields that you control in ME.

You can add airfields to two sets: the 'inclusion' and 'exclusion' sets. All civilian flights depart from and land at airfields from the 'inclusion' set that are not also on the 'exclusion' set. You can use this feature by manipulating the exclusion set to temporarily 'close down' airfields on the inclusion set, or to quickly remove just a few airfields in ME from the map for civilian traffic.

You add an airfield in ME to the inclusion list by adding the 'civAir' attribute to a trigger zone. The nearest operational airfield will then be added to the inclusion list. Note that if you have two trigger zones with the civAir attribute closest to the same airfield, this **airfield will be added twice, making it proportionally more likely to be picked** as departure or destination airfield. The value you supply to the civAir is disregarded – unless it is the word "exclude". If your civAir attribute's value is 'exclude' the nearest operational airfield is added to the 'exclusion' list instead. If an airfield exists on both 'inclusion' and 'exclusion' lists, the exclusion list takes precedence (no flight to or from that airfield).

DEPARTURE-ONLY/ARRIVAL-ONLY AIRFIELDS



Airfields can be designated as departure-only or arrival-only. To do so, simply use the keyword 'departure' (or 'depart only') or 'arrival' (or 'arrive only') as value to the civAir attribute. Airfields with a 'departure' value will not be used as landing airfields, while airfields with 'arrival' are not used for take-offs (or air-starts at mission begin).

Note that arrival-only and departure-only airfields are added to the list of generally open airfields before departure or arrival are determined. This means that if you add an airfield once as a generally permitted airfield, and then add the airfield again as 'arrive only', this airfield is still available for departure and landing; it's also more likely to be picked as arrival since its name is entered twice into that set. Use this to control the likelihood of airports being chosen as source or destination for flights.

IMPORTANT

If you add any airfields to the 'inclusion' list, it will contain *only those* airfields that you have added with zones. If you add no airfields to the 'inclusion' (or depart-only) list, CivAir adds *all of the map's airfields* to that list. Use the latter to have CivAir use all airfields, and then remove a few with the "civAir/exclude" attribute/value pair.

FLIGHTS
Unless the civilian flight is air-started (which only happens when enabled and only when the mission starts up), all flights start cold at ramp. Once they have started, the aircraft taxi normally, depart the airfield, overfly the destination airfield, and then land there. Once landed, they taxi to their parking lot, and power down. Some time later, they de-spawn, and a new flight is created.

An air-started (can only occur on mission start when enabled, used to populate the air at mission start) aircraft appear over their starting airfield, and immediately start flying towards their destination. If multiple aircraft air-start over the same airfield, they are altitude-separated to avoid a collision.

LIMITATIONS
Currently, civAir has the following limitations:

- Standard Airfields Only
  CivAir currently supports only standard airfields on your map. It does not support FARPs or Ships

- Fixed-Wing Only
  Currently, CivAir does not support helicopters.

3.5.8.2 Dependencies
CivAir requires dcsCommon and cfxZones

3.5.8.3 Module Configuration
civAir uses a configuration zone for customization. To customize settings,

- Place a Trigger Zone in ME anywhere
- Name it “civAirConfig” (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”
aircraftTypes	A comma-separated list of Types (as defined in https://github.com/mrSkortch/DCS-miscScripts/tree/master/ObjectDB/Aircraft) that define the aircrafts used for civilian flights. These must be fixed wing aircraft (i.e not helicopters). All airframe types are picked from this list, and each entry has the same chance to be picked. This means that if you list the same type twice, you increase the chance of that type to be picked. civAir defaults to the following list: • Yak-40, Yak-40, C-130, C-17A, IL-76MD, An-30M, An-26B Note that by default, the Yak-40 is twice as likely to be picked for a flight as an aircraft because its type is listed twice

Name	Description
ups	Number of updates per second that civAir checks on its flights. By default, this is 0.05, or once every 20 seconds.
maxTraffic	Maximum number of civilian flights at the same time. Defaults to 10
maxIdle	Number of seconds of an aircraft idling that can elapse before it is removed. CivAir determines that an aircraft is idling by checking if it is moving. If you set this number too low, a cold-starting aircraft may be removed before it can move. Defaults to 480 (seconds = 8 minutes)
initialAirSpawns	Controls if at mission start half of maxTraffic immediately spawn in mid-air to start a mission with planes in the air.

3.5.8.4 ME Attributes

In ME you can add airfields to the set of included or excluded civAir airfields. To do so, create a trigger zone close to the airfield, and add the following attribute:

Name	Description
civAir	When present, the airfield closest to this trigger zone is added to either the inlusion or exclusion set. Additional keywords are supported: • 'exclude' If the attribute's value is 'exclude' the closest airfield is added to the exclusion list. In all other cases, it is added to the inclusion list. • 'departure' or 'depart only' In this case, the airfield is added to the inclusion set only when looking for airfields to depart from • 'arrival' or 'arrive only' In this case, the airfield is added to the inclusion set only when looking for an airfield to land on. Note that you can add the same airfield multiple times by adding multiple trigger zones with a civAir attribute close to the same airfield. This will proportionally increase the likelihood of the airfield being picked for either destination or departure Note also that only functioning airfields are chosen. FARPS or ships are disregarded. **MANDATORY**

3.5.8.5 API

(tbc)

3.5.8.6 Using the module

Copy the civAir source into a DOSCRIPT action that runs at the start of the mission. No other change is needed, from that point on 10 civilian flights between two random airfields will constantly populate your map.

You can customize that setup in the above-described way.

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, OBSCURRED" 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	If true, all planes are automatically treated as actively reconnoitering. **NOTE** 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
redScouts	If true, all red planes are included as scouts when autoRRecon is true. Default is false
blueScouts	If true, all blue planes are included as scouts when autoRecon is true. Default is **true**
greyScouts	If true, all neutral planes are included as scouts when autoRecon is true. Default is false
playerOnlyRecon	If true, only player aircraft are included as scouts when autoRecon is true. AI planes will not be automatically included as scouts. **IMPORTANT** This condition is applied **in addition** to blueScouts and redScouts. If you disallow red scouts, red players will not automatically be added to the list of scouts. Defaults to false
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 e recon event occurs. Defaults to <nosound>, which will not play a sound
autoRemove	When a detected group is destroyed, that group’s mark is immediately removed from the map if this attribute is set to true Defaults to true

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-use: 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 **en**abled, 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.

Name	Description
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 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. **DO NOT CHANGE THIS UNLESS YOU ARE ABSOLUTERLY SURE YOU KNOW WHAT YOU ARE DOING**. Defaults to 0
enabledFlagValue	This reflects SSB's flag value of that same name. **DO NOT CHANGE THIS UNLESS YOU ARE ABSOLUTERLY SURE YOU KNOW WHAT YOU ARE DOING**. 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 unGrief (tbc)

3.5.13.1 Description

3.5.13.2 Dependencies

3.5.13.3 Module Configuration

3.5.13.4 ME Attributes

3.5.13.5 API

3.5.13.6 Using the module

3.5.14 cfxMon Development Tool (Lua Only)

3.5.14.1 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 automatic 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.

3.5.14.2 Dependencies

cfxMon requires dcsCommon and cfxZones.

Optional: all other DML modules

3.5.14.3 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.

3.5.14.4 ME Attributes
None.

3.5.14.5 API
None.

3.5.14.6 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.

3.5.15 Module Name

3.5.15.1 Description

3.5.15.2 Dependencies

3.5.15.3 Module Configuration

3.5.15.4 ME Attributes

3.5.15.5 API

3.5.15.6 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)
```

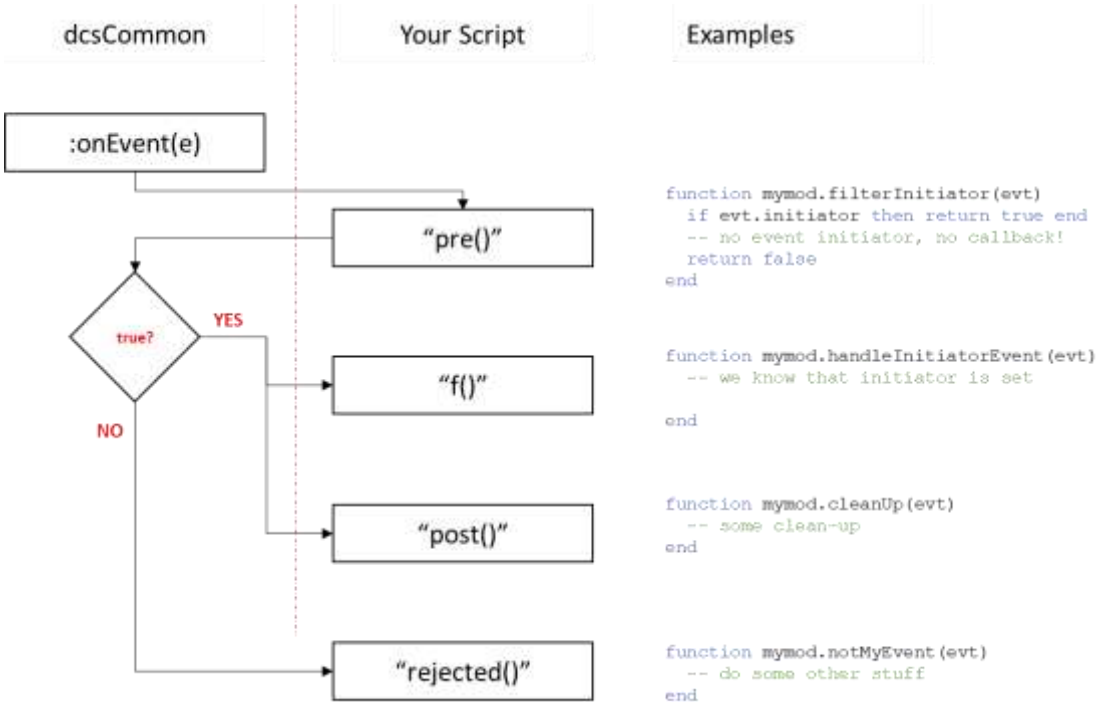
with

- `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 stings into arrays (for example a string containing a comma separated list if 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 sting 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, bur 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 bust 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.

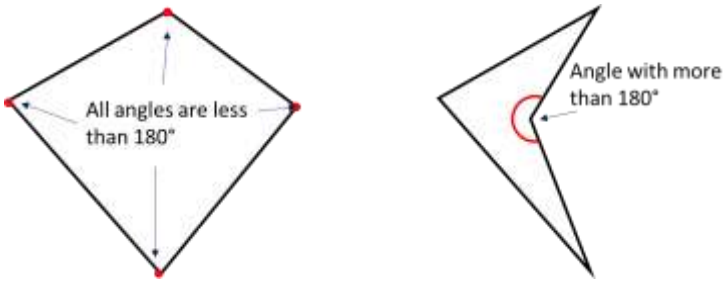
- **isCircle**
  A Boolean that is true if the zone is set up as a circle in ME, false otherwise.

- **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 a 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 fuill, 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 ("l10n/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 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 plance 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 cfxCommander hints on how to create paths that lead into, reside within, or lead out of this zone. Currently, the following hints are supported: • “normal” Create a path as requested • “offroad” Orders to move on roads are ignored, and off-road pathing is

Name	Description
	allowed. Use this in case an area has a complex road system that makes pathing difficult or degrades performance. **MANDATORY**

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

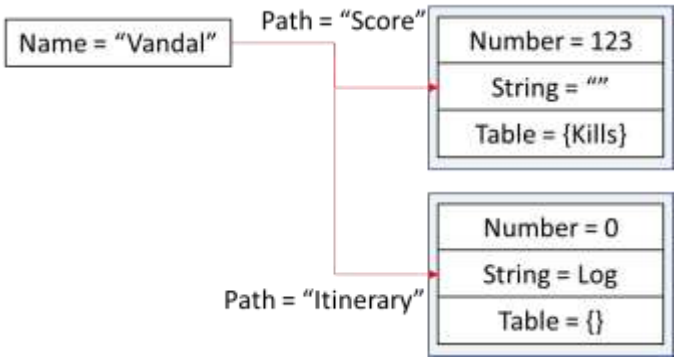
nameStats provides an easy to use, generalized, name-based information store. Mission designers first create a new store for a name (e.g. a player's name - "Vandal"). Once a store has been created for that name, data can be stored and retrieved inside that store by using 'paths' that lead to different 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 futher 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.

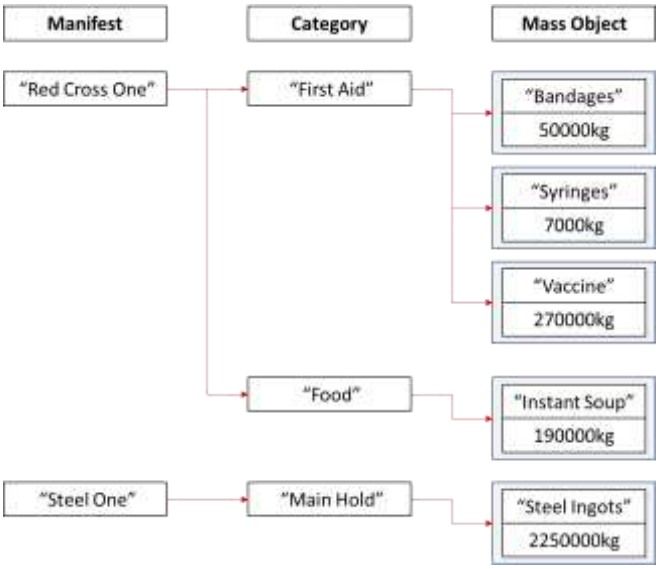
3.6.6.1 Description

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, tough, 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 units 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 gernerates 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 massObjct 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 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 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, catergory)

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 move 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.ups` – updates fer 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 theCargoObject. If the cargo object does not exist, nil is returned..

3.6.7.4.3 addCargo(theCargoObject)

Adds 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 objected with the name cargoName from the pool of watched cargo objects.

3.6.7.4.5 removeCargo(theCargoObject)

removes 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 them 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
queuedUpdates	set to true to process one group per turn. To work this way, scheduledUpdates (see below) must be false. Default is false
scheduledUpdates	set to false to allow queued of standard updates. Overrides queuedUpdates is true. Defaults to false
maxManagedTroops	Defines a cap on the number of actively managed groups (*not* 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 **-1** for an infinite cap. Defaults to 65

Name	Description
monitorNumbers	Set to true do receive debugging info when queuing and de-queuing 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 fram 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 tbc (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

3.6.10 cfxMX tbc (Lua Only)

3.6.10.1 Description

This low-lever module primarily supplies low-level information about groups as defined by the Mission in ME. It is primarily used to clone groups that are present at mission start

3.6.10.2 Dependencies

3.6.10.3 Module Configuration

3.6.10.4 ME Attributes

3.6.10.5 API

3.6.10.6 Using the module

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

```
dcsCommon.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 = dcsCommon.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 topClone(orig)

Creates a “top clone” from orig: the first level of the table are all newly created, any objects they reference, however, are copied by reference. Sometimes also called a ‘shallow clone’

4.1.2.5 clone(orig)

Recursively creates a clone from orig. All referenced objects are also cloned, creating a true clone.

4.1.2.6 copyArray(inArray)

Creates a new array that references all elements of inArray in exactly the same order

4.1.2.7 enumerateTable(theTable)

converts the arbitrary array/table theTable to an array, dropping any keys.

4.1.2.8 arrayContainsString(theArray, theString)

returns true if the array theArray contains an element that is equal to theString

4.1.2.9 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

4.1.2.10 trim(inputstr)

Returns the contents of inputstr with leading and trailing blanks (“ “) removed.

4.1.2.11 trimArray(theArray)
Returns an array based on theArray, in which all elements have their leading and trailing blanks (“ “) removed.

4.1.2.12 stringStartsWith(theString, thePrefix)
Returns true if theString starts with thePrefix, false otherwise

4.1.2.13 removePrefix(theString, thePrefix)
Returns the contents of theString with thePrefix removed. If theString does not start with thePrefix. theString is returned unchanged.

4.1.2.14 stringEndsWith(theString, theEnding)
Returns true if theString ends with theending, false otherwise

4.1.2.15 removeEnding(theString, theEnding)
Returns the contents of theString with theEnding removed. If theString does not end with theEnding. theString is returned unchanged.

4.1.2.16 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.

4.1.2.17 numberUUID()
Returns a unique number for each time it is invoked

4.1.2.18 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 parkingslot 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 rage 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 lining 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 createOverheadAirdromeRoutPintData(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 dy, 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 dy 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. thUnit 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 removeEventHanlder

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 removeEventHanlder

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 the 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 they 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 `linkUnitZoZone()` 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 `zonesStrtingWithName()`.

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 retuned 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 getPositiveRangeFromZoneProperty(theZone, theProperty, default)
Interprets the value of the property “theProperty” as a hyphenated range of positive numbers (e.g. “12-34”). Returns two values: lower bounds and upperbounds.(12, 34). If the numbers in the range are given in the wrong order (higher value, lower value) the returned values are sorted correctly (lower, higher)

If only one number is given (e.g. “23”), it returns that number for lower and upper bounds.

4.3.3.14 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.

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 (NDB)**
  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.

- **ME-Triggered Spawns (Spawner)**
  Shows how **ME triggers** can be used to **cause a Spawner to spawn**

- **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

- **Random Glory / Random Death (RND Flag)**
  Demonstrates the use of rndFlags modules to **randomize control flow** in your mission. While Random Glory also somewhat shows off DML- DML integration, Random Death shows the **classic** '**Randomized Enemies**' setup, enormously simplified with a rndFlags module.

- **Pulsing Fun (Pulser)**
  Shows how to use a pulse module to **drive / stagger actions (via flag changes).** Like Random Glory, it also shows off DML-DML integration.

- **Attack of the CloneZ (Clone Zone)**
  A mission that shows but a few of **the many great ways to use clone zones**: to save time, to randomize, to provide endless clone spawners etc. Also demonstrates

some of the many uses of delayFlags

- **Flag Fun Zone-Local Flags**
  A mission that show how you can use **zone-local flags** to build simple zones with modules that exchange information internally, and **then use copy/paste** to quickly deploy lots of self-contained copies

- **Once, twice, three times a maybe (Count Down)**
  A mission that uses a **counter to control how many times a cloner spawns**, and then starts some smoke when the counter has counted down and the cloner no longer spawns

- **Bottled Messages (Messenger)**
  Introducing: **messenger** and **delayTimer.** And **compound module zones**. This mission also shows how to initiate DML-actions with ME triggers

- **Follow Me! (Unit Zones, and Messenger)**
  Shows how to use DML's version of the good old GROUP IN ZONE ME flag – except that DML may be way cooler. You decide.

- **Clone Relations (Advanced Clone Topics)**
  Shows how a cloner works with templates that include escort targets and other template-external targets and how it automatically resolves this.

- **Moving Spawners (Spawn Zones and Linked Units)**
  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) – Cargo Receiver**
  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

- **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 (Owned Zones)**
  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.

- **FARP and away**
  A very simple demo that shows how a **FARP zone is used** to set up some defenders, and how it react to capture. Again, a show only, no flying required.

- **Keeping the score (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.

- **The Zonal Countdown (Count Down, Messenger)**
  Countdown, Messenger and **continuous updates**. With a bang at the end. Also shows **multiple flag triggering**

- **(Full Mission) Frog Men Training**
  This mission **brings together several modules and ME Flags** for a real-world weapons training mission. ME Flags are used to enable modules, and DML is used for missile protection, re-spawning, repair/restock, and messaging.

- **CSAR of Georgia (CSAR Manager)**
  Demonstrates **how to set up and use the CSAR Manager** package in your missions.

- **Track This! (GroupTracker)**
  The basics of the **groupTracker** module and how it's various **output flags** can be used in your mission to greatly simplify otherwise complex tasks.

- **Watchflags Demo**
  A mission that demonstrates the various **capabilities all input flags possess** and how to activate them.

- **Viper with a double you (Wiper)**
  Removing objects – the good, the bad, and the ugly.

- **Radio Go-Go**
  Using the Radio Item → DML module flag glue to trigger a module multiple times with the same Radio Item

- **(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.

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.

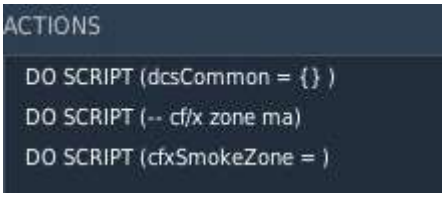


The smoke on the right, however, keeps happily on smoking. Oooh, rah, score one for DML! Yes, not that impressive, but let's move on to ME

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]*!



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.

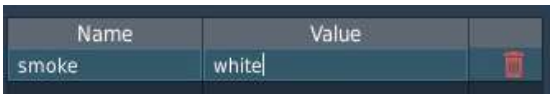
5.2.3 Discussion

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?

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

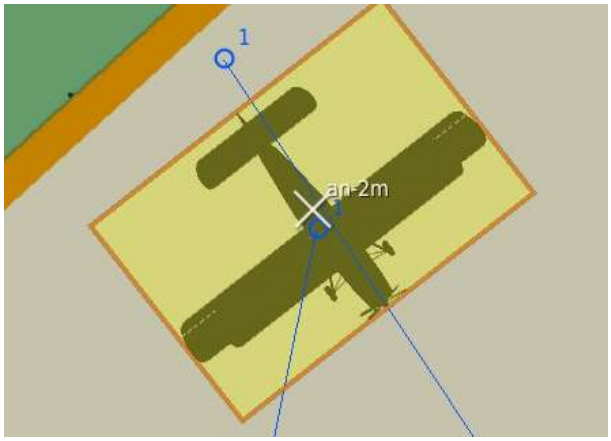
Start the mission either as Su-27T (free with DCS) or UH-1 (not free). Then destroy the An-2M (the double-decker that is part of the scenery at Senaki-Kolkhi):

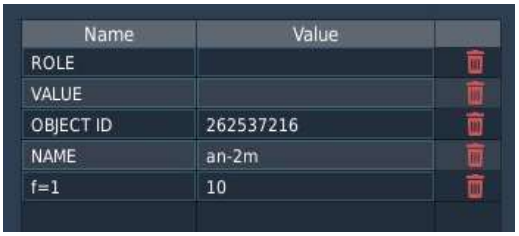


As soon as it is destroyed, you will see a message appear to that effect. Note that DML doesn't care how you destroy the Antonov. Try being creative 😊.

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.

Note:
This demo's documentation uses the old, deprecated '*f=1*' attribute. The (updated) demo itself uses the newer *f!* (bang!) attribute with method = "flip".

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

5.4.1 Demonstration Goals

Shows how NDB zones can be used to place beacons all over the map. We show in this mission

- How to set up NDB for KHz and MHz ranges
- How to place moving NDB
- An ELT locator set up as NDB

Note:
To see in-game how these zone enhancements work, you need

- An **aircraft with ADF** functionality built in (e.g. Huey, F/A-18, F-5E). The SU-25T has no ADF capability, and hence can't be used. The mission has a Huey and Hornet included.
- You need to **know how to use ADF and operate the radios**. It's outside the scope of this document to teach you how. Simply accept the premise that it works if you currently lack the modules or radio navigation know-how





5.4.2 What To Explore

5.4.2.1 In Mission

Enter the Huey or Hornet there are three NDB in the KHz range. Use your ADF to locate them:

- 1100 KHz (fixed, at end of runway)
- 540 KHz (moving, on the battle cruiser off the coast to the west)
- 420 KHz (moving with the Hummer along the far runway side)

There are also two NDB in the MHz range

- 121.5 MHz (fixed, emergency frequency), in the Batumi Harbor
- 125 MHz (fixed, at the beginning of runway 24)

If you have access to an ADF that operates in the KHz range (Huey), tune to 420 KHz, and observe how the needle slowly 'walks' from SE to W, following the Hummer. Now tune into 125 MHz and see ADF point to the beginning of runway 24.

Now, make sure that you hear the warbling "distress beacon" audio (which we incorrectly use for all NDB - you wouldn't use that sound for "normal" NDB) of either NDB 420 KHz or 125 MHz.



Carefully taxi your aircraft to somewhere inside the "NDB 420/125 OFF" zone (marked in purple on the right).

Notice how the warble sound cuts out, being replaced with static, and the ADF indicates that it has no source.

Now taxi to a spot inside the "NDB 420/125 ON" zone at the start of runway 24. Note how the NDB comes online again, the audio starts screaming, and the ADF indicator points to the selected NDB.

Now take off, and try to locate the cruiser, using her 540 KHz beacon to home in.



Now switch to the Hornet and locate the ADF at the start of the runway. Then take off, and home in on the 121.5 ELT signal.

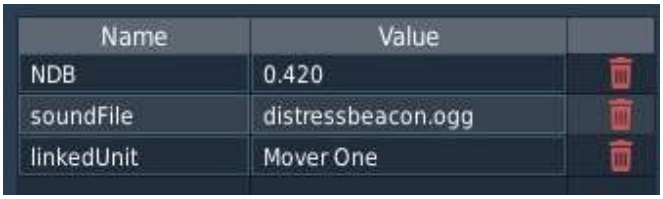
5.4.2.2 ME

Note how placing NDB is quite simple. To have an NDB follow a unit, simply put its name in the linkedUnit attribute.

Note that linkedUnit is not an attribute from the cfxNDB module, but it is inherited from cfxZones. Remember that you do not have to place a zone that follows a unit exactly, it will center itself on mission start. Here, we've placed it merely close to the unit it follows to make it visually easy to work on both without obstructing each other.



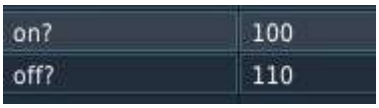
Note that although an NDB's frequency is defined in MHz, creating an NDB in the KHz range is easy: divide the KHz by 1000 and enter that number.



Example: to create an NDB at 420 KHZ, put 420/1000 = 0.420 into the NDB attribute.

Using ME Flags to switch NDB on and off

See how we use ME flags 100 and 110 to turn on multiple NDB at once by using the "on?" and "off?" attributes in NDB 420 KHz (the one following the Hummer) and NDB 125 MHz (the one in front of runway 24). Note that multiple NDB can share the same trigger flag.

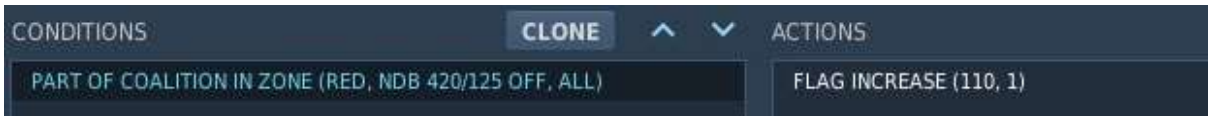


5.4.3 Discussion

Again, no Lua at all.

The ability to place NDB on ships and moving vehicles makes it possible to easily create some nice dynamic 'locate the hidden transmitter' missions.

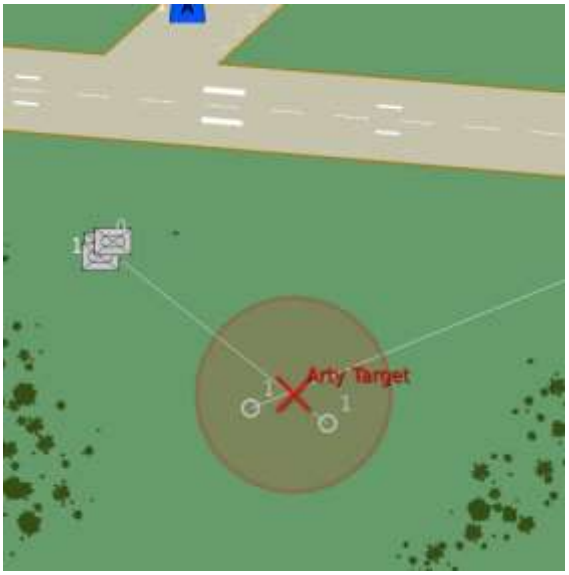
Being able to start and stop NDBs with triggers also allows some sneaky mission surprises, for example simulating ELT failure in the last moment when you are just about to discover the unit that you are looking for.



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"

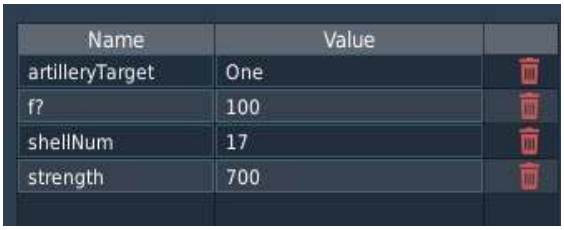
Notice that almost instantaneously, shells explode around those vehicles.

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

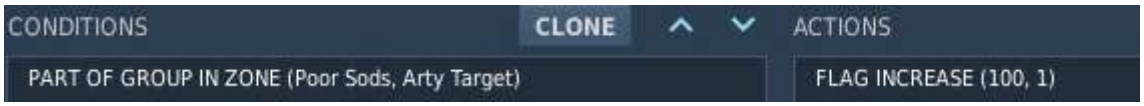
5.5.2.2 ME

Look at the trigger Zone “Arty Target”. This is the target zone that is triggered multiple times. First, look at the attributes:



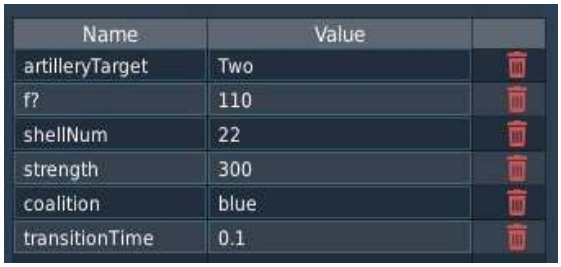
Name	Value
artilleryTarget	One
f?	100
shellNum	17
strength	700

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)



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:



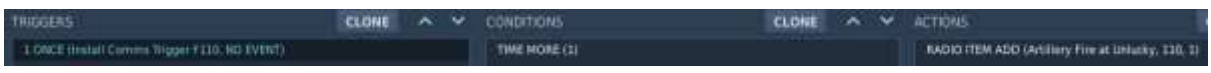
Name	Value
artilleryTarget	Two
f?	110
shellNum	22
strength	300
coalition	blue
transitionTime	0.1

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.

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



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.

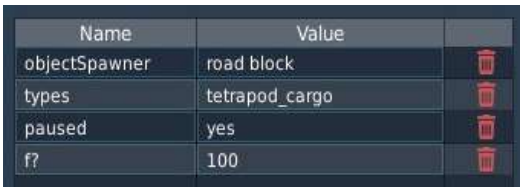
5.6.2 What To Explore

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.



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)

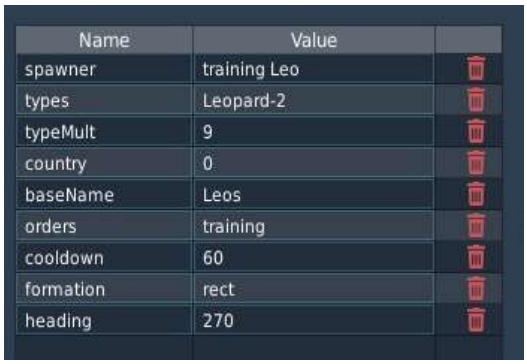
Destroy the red vehicles (you have unlimited ammo). After you destroy a group of ground vehicles, they will re-spawn (after a delay of 60 seconds). Respawns are unlimited

Now destroy the JTAC group. They won't respawn.

Finally, use F7 to cycle though 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

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), bit 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 Random Glory / Random Death (rndFlag)

5.8.1 Demonstration Goals

The module to kill for. Yes, *finally* a way to simply and efficiently randomize missions. In more ways than you can shake a bag of dice at.



Although not technically a Zone Enhancement (it pretty much ignores everything that makes a trigger zone a, well, trigger zone - we simply use a Zone so we can pass data to the module with ME) we know that it will soon become one of your most-often used module.

In “Random Glory” we use the rndFlags in a benign manner: to randomly select and activate a bunch of smoke and object spawners.

In your real missions, however, your objectives are going to be more sinister: you’d be choosing and activating groups of enemies, all armed to the teeth and intent on killing the hapless players. Hence “Random Death” – a mission that randomly chooses a couple of deadly enemies from a bunch of possible opponents that will hunt you down. See if you can complete your mission (destroy the tanks marked as primary) and live to tell. Then try to do it again.

5.8.2 What To Explore

5.8.2.1 In Mission

Random Glory

This is a kind of interactive show. Start the mission, enter the Frogfoot, and relax. Look outside and notice that nothing is happening. Placed on the map in front of you are a number of smoke zones, and a number of (static) object spawners. None of them is active yet.

Go to communications→other.. and notice that there are two strange menu items “Change RND Watchflag to 1” and “Change RND Watchflag to 10”. Choose one of them and wait for a second or so. Depending on, well, some random numbers, some smoke will start, and some F-117 may appear in front of you.

Now choose the other menu item. More planes and colorful smoke. Continue to alternate between the two menus and more smoke and planes appear until finally, a C-17 (big, big) cargo plane appears to the left.



That's all, show is over.

(if you continue changing the watchflag, you can marvel at the magnificent C-17 stacking. It's a result of how I wired the 'done+1' output that fires the object spawner. Sosumi).

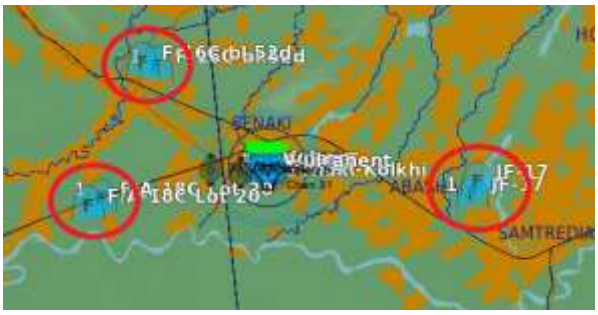
Random Death

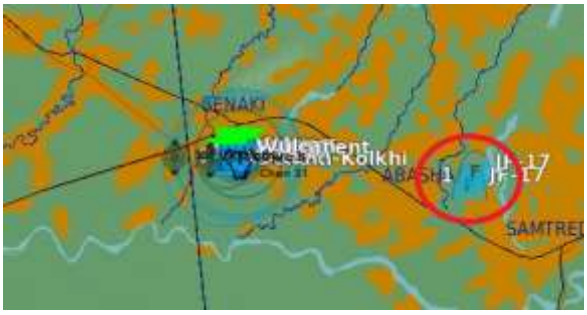
Start the mission in your Frogfoot. Look at the map. Your targets are the vehicles close to the tent on Senaki-Kolkhi's tarmac (marked with a red circle). Destroy them and you win.



A few seconds after the mission starts, you see a message "Here we go!", along with some strange chatter running along the right side of the screen. Look at the map again and note that new blue (enemy) units have appeared. Note what has appeared, and try to complete the mission if you are so inclined.

Re-start the mission. After you get the "Here we go!" message, look at the map again





You'll notice that a different number of enemies, and quite probably also different enemies have appeared. Re-start once more. Again, there'll be a different mix of enemy units.

5.8.2.2 ME

Random Glory

In ME we see a couple of interesting things. For experienced mission designers the most peculiar aspect is that there is only a single trigger, the one that loads DML and sets up the menus. **Everything else doesn't need ME trigger actions**!

So how does it work? DML modules have strong ME integration, which means that they can either watch flags for changes to trigger their action, or can set flags as their action. Which means that if we connect these two directly with the flag in the middle, we can completely cut ME's Triggers out of the middle: the random generator creates flag changes, and the smoke and spawn zones watch for changes

In this mission we first set up a bunch of smoke and spawn zones, and key them to different flags to activate. For example, Smoke Zone ‘smoky-1’ is keyed to activate when flag 1 changes, while ‘smoky-2’ activates when flag 2 changes etc. In total we have ten some zones, each waiting for their own flag to change: flags 1 through 10.



We then added some object spawners, that also wait for a flag’s change to spawn. These spawners look at flags 30 through 32 for a change.

To prevent all zones from doing anything before their watched flags change, they are all set to ‘paused’.

Finally, we added a randomizer that each time it’s watch flag (f?=100) changes

Name	Value
RND	random gen
flags!	1 - 10, 30-32
pollSize	1-3
remove	yes
f?	100
done+1	110

- picks one to three flags (pollSize 1-3)
- out of the set of 13 flags (1-10, 30-32)
- and sets them to 1 (method on, set by default)
- afterwards, the flags that were drawn, are discarded (remove = yes)
- When all flags have been drawn from the set, and a cycle is requested on an empty set, the randomizer increases the value of flag 110 by one (done? 110)

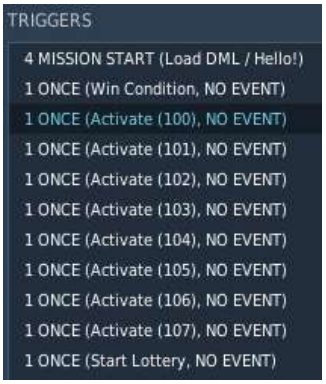
When you inspect the start trigger that loads DML, there are also two actions that connect the menu items “Change RND Watchflag” and that change flag 100 to 1 and 10, respectively. This setup is required because the randomizer looks for a change in a flag, and menu items (unlike DML) can’t simply increase a flag, they can only set a flag to a fixed value.



So if we want to trigger the randomizer multiple times, we have to set flag 100 to different values: 1 and 10. Each time you set it to a different value, the randomizer fires, and some more smoke or airplanes appear… Until the last flag is removed from the set of flags. If the randomizer is cycled once again, it changed the done? flag (110) instead. That one is ‘connected’ to the spawner for the C-17. So that plane appears when all flags have been exhausted. And keeps appearing, stacking them on top of each other, when you keep changing the randomizer’s watchflag.

Random Death

In contrast to ‘Random Glory’, this mission probably looks much more like you would have expected, and it works exactly how you imagined it: we set up a bunch of triggers in ME, each one to enable a group of enemies. We take these triggers and feed them as a set of trigger into our randomizer.



At some point in the mission (when the Frogfoot leaves the start circle, which triggers ‘Start Lottery’)) that randomizer picks a couple of them by random, and set’s their flags to true – causing the units to become active and begin their hunt for you.

The randomizer’s setup itself also bears no surprises: It uses the set of flags as source set (100-108, a total of 8 items). It’s paused at mission begin (no atStart attribute given), and waits for flag 99 to change for a cycle. We only plan one cycle, and this is simply wired up with a ‘ONCE’ ME trigger (Start Lottery) that fires when the Frogfoot leaves the starting circle, setting flag 99 to 1.

Name	Value
RND	select enemies
flags!	100-107
pollSize	3-5
f?	99

When that happens, the randomizer initiates a cycle (f?=99): it rolls the dice on the number of flags to change (pollSize 3-5), and then randomly picks that many flags from the pool (100-107, 8 total) to change.

Since method is implicitly set to ‘on’ (chosen by default), those flags are all set to 1, which fires their triggers in ME, enabling their groups. And presto! you have a bunch of deadly enemies heading your way.

5.8.3 Discussion

Again, none of these missions require any Lua.

‘Random Glory’ demonstrated how DML modules use flags to communicate with each other without needing any ME mediation. They exchange signals with each other and you, the mission designer, don’t have to set up any additional trigger actions.

That being said, the randomizers can randomize a mission much easier, even using a classic (non-DML) approach with triggers and actions that are then driven by randomizers.



So, just **how effective is a randomizer** to make our mission more unpredictable? Well, let’s run the numbers to find out. We have 8 different groups (objects), and we can draw 3, 4 or 5 samples from the 8 groups. Elementary statistics gives us the number of different combinations for each draw size at 56 combinations for 5, 70 combinations when we draw 4, and again 56 different combinations when we draw 3 out of 8 items. That makes for a grand total of **182 different combinations** (don’t believe me? Here’s how we calculated that number. $C(n,r) = \binom{n}{r} = \frac{n!}{r!(n-r)!}$ with n = 8 and r = 3, 4, and 5)

5.9 Pulsing Fun.miz (pulseFlags)

5.9.1 Demonstration Goals

Show how a pulse module (“Pulser”) is used to repeatedly initiate flag-driven actions. This is useful for all kind of controls in your mission: staging of waves, initiating actions after a certain delay, synchronizing and/or randomizing event etc. In Pulsing Fun we use it for a simple, literally earth-shaking purpose. to simulate an artillery barrage.

5.9.2 What To Explore

5.9.2.1 In Mission

Enter the Frogfoot. Be sure not to move it an inch, or you’ll die. Open a beverage, and enjoy the show. Change to F2 outside view for an even more impressive display of destructive fun.



5.9.2.2 ME

This mission (similar to “Random Glory”) demonstrates the stark difference between DML and classic ME mission crafting: MW works with Triggers, Conditions and Actions. DML replaces the Conditions with modules, and allows you to directly wire modules together with flags, which they use as triggers. ME’s method is much broader in scope. DML is much easier to use.

In this demo, the pulser regularly kicks the Randomizer into action via flag 99. The time between the kicks is randomized between 1-3 seconds, and repeats forever.

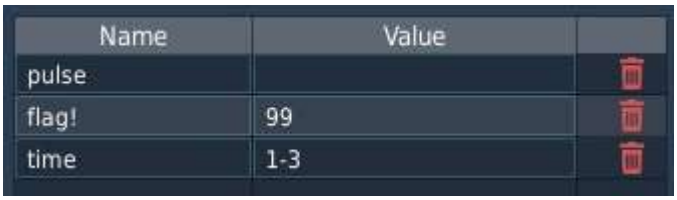


Each time the randomizer receives a kick, it picks 1 to 3 flags from it’s pool of flags 200-210, and flips those flags it has picked

Th artillery zones all listen to their flags (200-210), and whenever its value changes, simulate a shell landing in their zone. This cascades nicely, completely unpredictable, into a great show. And no ME Triggers were hurt in this demo!

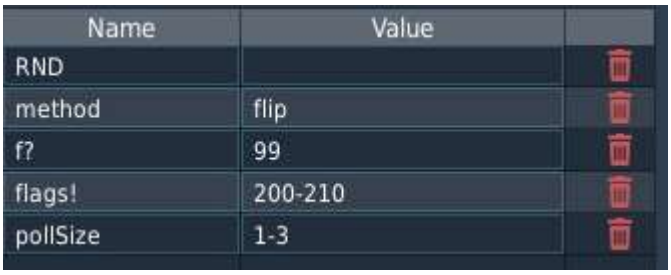
5.9.3 Discussion

Again, no Lua. Furthermore, we don't even use triggers (except for loading DML).



The various modules use flags to signal to each other, completely eschewing traditional ME Trigger/Action combos.

Since DML triggers are looking for a value change in the flag that they are watching to trigger, and we expect to trigger the artillery zones multiple times, we use the 'flip' method. This method alternates the flag's value between 0 and 1, neatly generating a signal that the modules can see.



We can do this because we know that the randomizer only drives other DML modules, and the flip method is therefore the most elegant, guaranteed to never create an under/over flow situation while still registering a change. Ok, it would probably have worked just as well with increment, but it feels let neat. It won't work with either 'on' or 'off' because these methods set a fixed value, and on all subsequent triggers, the value does not change on the flag, keeping the artillery zone quiet.

Although this demo is a bit contrived (yet impressive!), it serves nicely how you can use a pulser to stagger starting stages randomly. Use this and similar cascades to create much more varied missions, with less risk of repetitive play-through.

5.10 Attack of the CloneZ.miz (Clone Zones and Delay)

5.10.1 Demonstration Goals

This mission demonstrates how you can use clones to randomize missions, and quickly (with very little effort) deploy units from spawners – again randomized.



This mission focusses on several aspects:

- How to create templates that other zones can use
- How cloners can use multiple ‘sources’ to randomize what they spawn
- How clone spawning can be randomized with rndFlag randomizers
- How cloners can be used to provide “endless” spawns
- How to use a delayFlag to pause respawns for a while
- How cloners can be used to cascade attackers until they overwhelm the defenders

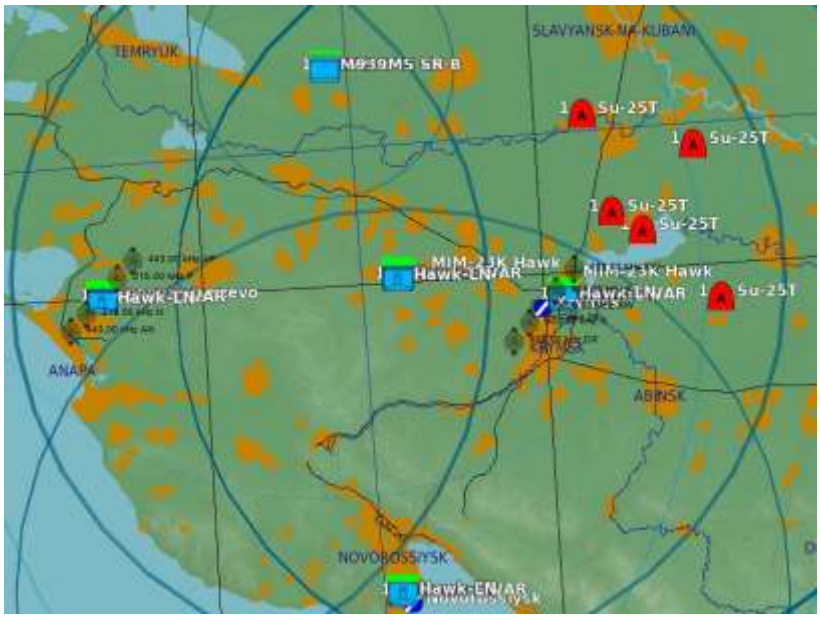
5.10.2 What To Explore

5.10.2.1 In Mission

This is another mission where you mostly observe, and we’ll then run through the same in ME to see how we did it. So, start the mission, choose either side, and hop into your Su-25T. Don’t touch the controls and go straight into F10 Map view. Zoom out and pan the map far up northwest, until you can see Anapa, Novorossiysk, Krymsk and the single red aircraft approaching Krymsk at the same time. Pause the game.

Let’s take a good look around. Take note of the SAM locations north-west of the Krymsk – Novorossiysk line. Inspect them and note their size and composition. Now end the mission, and choose ‘Fly Again’ and repeat the procedure. Notice that not only have the number and locations of SAM sites changed, so has their composition. Try this some more times. You’ll find that there are four locations in total, and that each location can either be populated with a SAM or not. Furthermore, a SAM’s composition can vary between three types: a small HAWK battery, a large HAWK battery and a NASAM battery.

Now return your attention to the lone Su-25T that is approaching Krymsk. Un-pause the game (if you followed my instructions), and watch it valiantly attack Krymsk's runway. It won't succeed, the SAM battery will see to that. Note that once the Su-25T is shot down, three more Su-25T, all identical to the first appear: one where the first originated, one to the northwest, one to the southwest. They all approach Krymsk, trying to bomb the runway. They all will initially fail. Note, however, that as soon as the first Su-25T of the second batch is shot down, three more appear. This continues until the SAM battery is out of missiles, and no more Su-25T are shot down. This can briefly lead to situations, where five or more SU-25T are attacking Krymsk.



Now restart the mission, and pan the map to Gelendzhik and zoom in. The harbor area is protected by 8 T-90. Some 5 seconds after the mission starts, you'll see a group of three Hummers spawn to the east, approaching the Harbor. They'll never make it. A few seconds after they are destroyed, a new blue group of three Hummers spawns – maybe from the same location, maybe from one of the other two possible locations.



Simply observe as group after blue group spawns from one of three locations. If your sense of timing is acute, you will also notice that the interval between one blue group being destroyed, and a re-spawn appearing varies by a few seconds.

Finally, pan down to beautiful Batumi and look at Batumi airfield's tarmac. Note the many blue units on the tarmac. Enter the blue Su-25T (if you aren't already in it), go to F2 outside view, and inspect them. Eagle-eyed observer that you are, you will surely have noticed that there is something repetitive about them - they seem to be four identical sets of troops.

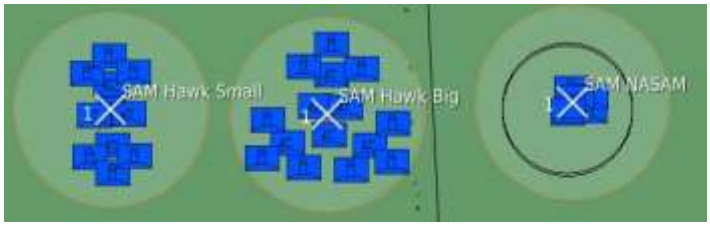


Nothing left to see, let's fire up ME and have a look at what's working behind the scenes.

5.10.2.2 ME

RANDOM SAMS

Let's begin with the part that wasn't visible in the mission. Far up in the North, there are three groups of blue SAM sites.



These are the SAM sites that you met in the mission: a small HAWK, a large HAWK, and a NASAM. Each of these SAM sites is inside their own Trigger Zone. If you inspect the trigger zones, you'll find that they are all alike: they only have a single attribute "cloner", and nothing else.

If you consult the documentation for Clone Zones you'll find that such a clone zone does the following: it creates a cloning template, and then, at mission start, removes all units. Since there are no instructions to spawn any clones (no 'onStart' or 'spawn?' attributes), these clone zones will never spawn any clones during a mission. If you were to run the mission and pan up to the North, you'll find nothing, the SAMs aren't there.

Name	Value
cloner	big sam hawk

We simply use a convenient, yet far-away location on the map to create our templates that we then use all over the map. Using out-of-way locations like this is convenient if you later want change their composition without the risk to accidentally change other units.

Now scroll south and to the east where you *did* see the SAM sites during the mission. In ME; there's nothing there except a few empty Trigger Zones with names like "Krasnyy SAM" and similar. So, let's inspect their attributes. These Trigger Zones are also clone zones. Unlike the first three, these *do* have a "source" attribute, meaning that they import a 'foreign' template. Strangely enough, the 'source' attribute doesn't list just one Trigger Zone as template, but three. This is a native randomizing feature of Clone Zones: when you provide more than one template, each time the zone is told to spawn clones, it randomly chooses one of the named template zones. That explains why all SAMs could have one of three different compositions each play-though.

Name	Value
cloner	
source	SAM NASAM, SAM Hawk Sma
spawn?	11

But how did we get a different number of SAM sites to spawn each play-through? The answer lies in the clone zones' *spawn?* flag, and the fabulous randomizer "CHOOSE RANDOM SAMS' zone east of Anapa. If you look at the randomizer's attributes you'll see that it randomizes once, at mission start (*onStart = yes*). So, when the mission starts up, the randomizer first chooses a *pollSize* (two to four) meaning that it will populate two to four of the existing SAM clone zones. Each SAM clone zone has their own *spawn?* flag to watch (from 10 to 13), and when that flag gets changed by the randomizer, that clone zone pulls a

Name	Value
RND	pick blue sam SITES
pollSize	2-4
flags!	10-13
onStart	yes

random template (from the three it can choose from), and clones it. So each mission you have two to four SAM sites, each populated with one of three possible SAM templates. That's more than 30 ways your blue SAMs are set up using only one randomizer and three templates.

WAVES OF CLONED ATTACK PLANES

Take a look at the Su-25T NE of Krymsk that is set up perform a Runway Attack. In addition to its initial location, its route has two waypoints: one for line-up and one to position it overhead Krymsk. Superimposed over the attack plane we find another "Red Clone One" Trigger Zone that functions as a clone zone.

Since this clone zone has no *source* attribute, any units inside function as template, so the SU-25T with orders to bomb Krymsk's runway will be cloned.

Name	Value
cloner	
empty+1	100
spawn?	100
onStart	yes

This cloner has an *onStart = yes* attribute, meaning that when the mission starts, the **template is created, and immediately afterwards, a clone is spawned** into the mission. This is markedly different from the SAM templates that we looked at before – this time the contents of a clone zone are immediately available in the game.

We also see with *spawn?* that every time flag 100 changes, this cloner will undergo a new spawn cycle. We also see that, strangely, *empty+1* is also queued to flag 100, meaning that every time the last spawned clone is destroyed, flag 100 changes. This creates a feedback loop: when the last unit of the last cloned batch dies, a new spawn cycle is initiated. This will continue until the cloned group is no longer completely destroyed.

And now for the cool wrinkle: "Red Clone Two" and "Red Clone Three". These are cloners that look very similar to "Red Clone One" – with two exceptions:

Name	Value
cloner	
source	Red Clone One
empty+1	100
spawn?	100

- They use Red Clone One as template, meaning that they spawn exact copies of whatever the template for Red Zone One is
- They do not have an *onStart = yes* attribute, meaning that when the mission starts, they don't spawn.

And there is something important that the cloners have in common, which we use for a surprise effect: they also *spawn?* and *empty+1* on flag 100, creating a triple feedback loop. This means that **whenever one of the clone groups is destroyed, all three re-spawn**. Be careful when using feedback loops, as they can create a big number of units in a very short time (even exponential growth). See this mission's Discussion section on how we limit spawning in this mission to ensure that this does not happen.



Taken together, this explains the initial mission behavior: A lone SU-25T that is followed by wave after wave of three SU-25T that originate from three different locations all at once, with a growing number of planes attacking Krymsk

DELAYED ENDLESS RANDOM RESPAWNS

Now let's look at Gelendzhik. Here we observed that the Hummers spawn in three different locations a short while after the previous spawn was destroyed. The set-up is similar to the three planes that we looked at before:

Name	Value
cloner	blue poor basterds
spawn?	210
empty+1	200

One group that defines the template, and two more groups that reference the "blue meanie" template with their *source* attribute. Unlike before, the "blue meanie" template does not spawn on start, and waits for a signal on flag *spawn? = 210* to spawn. We also notice that if the last unit of the cloned group is destroyed, flag 200 is changed via *empty+1*.

When we examine the other cloners, they look very similar, and just like the planes above, they have both important similarities and changes:

They have in common:

- Neither has *onStart*, so they all wait for a signal on their *spawn?* input

Name	Value
cloner	blue poor basterds
source	blue meanies
empty+1	200
spawn?	211

They are different

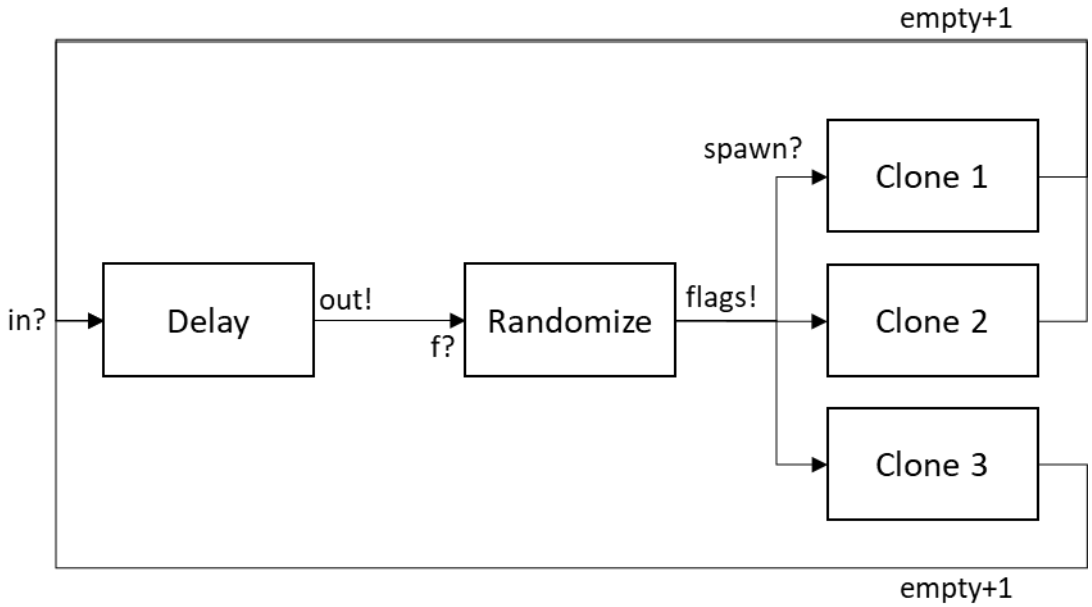
- All *spawn?* values are different, meaning that they all require different flags to change to spawn
- Unlike "blue meanie" that spawns its own template, these spawners use a foreign template (blue meanies', to be exact)

This setup is reminiscent of how we set up the randomized SAMs, and indeed we use a randomizer to pick which group to spawn. But if we fed *empty+1* directly into the randomizer, the new group would spawn immediately after the last was destroyed. That's where the delayFlag zone comes in. It watches the input flag *in?*, which is set to 200 – the flag that all blue meanie clones set when they are destroyed. So, when a group is destroyed, that flag changes, which initiates delayFlag's timer. When that timer runs down (here a random delay of 5 to 7 seconds) it bangs *out! = 201*, which feeds into the randomizer's *f? = 201* input trigger.

Name	Value
timeDelay	5-7
in?	200
out!	201

The randomizer, set to a fixed *pollSize* of 1 (implicitly, as no attribute means that a default of 1 is chosen automatically) selects one of the three output *flags!* (which in turn each connects to one of the three blue meanie clone zones), completing the circle – a new group is spawned.

Name	Value
RND	New blue after kill
flags!	210-212
f?	201
method	inc



This loop randomly spawns a new clone group, after a delay, when the previously cloned group was destroyed.

And a final note: the delayFlag zone also has an *onStart = yes* attribute. This starts the entire cycle when the mission starts.

MULTI-GROUP TEMPLATES

Finally, we turn our attention to Batumi. After we have discussed everything above, there's only one small thing left to explore: Multi-group templates – or rather: the fact that a template does not care if you give it one or many groups into the template. It simply copies, thank you sir!



If you inspect the "Two Group Template", you'll notice that it's a standard clone template setup: simply copy, then serve as template (no spawn by itself). If you look closer at what it clones, you'll find that there are actually *two* groups inside the clone zone: one consisting of infantry, and one consisting of a Hummer. While – obviously – such a setup can be streamlined in ME into a single ground forces group, such a setup can serve multiple purposes:

- This way, separate groups can have different routes even though they are spawned at the same time. Since routes are on a group level, you must use multiple groups if you want to assign different routes to parts of the spawn
- If you are using Helo Troops, it may be advantageous to split infantry and vehicles, because Helo Troops refuses to pick up groups that include vehicles

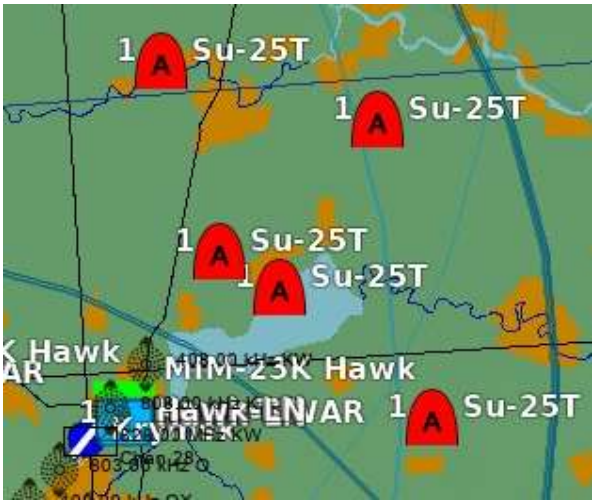
- More importantly, though, cloners aren't just restricted to ground forces. This way you can create a clone template that includes **a mix of ground forces, aircraft, helicopters and/or ships**.

Looking even closer, you'll notice that there's also a static object inside the clone zone. Static objects also become part of the template and will be cloned just like units. Be advised that in order to pass a an empty+1 check, all static objects must also be destroyed if they are included in a template.

5.10.3 Discussion

Lua? What Lua?

"EXPONENTIAL GROWTH"? NOPE!



Above, I ominously wrote "Be careful when using feedback loops, as they can create a big number of units in a very short time (even exponential growth)". While this is true with spawners, and you *can* easily create situations where the number of units cloned doubles every second (giving you less than 16 seconds to respond before DCS throws in the towel), there is a throttling factor built into this scenario.

The setup is that when a cloned group is destroyed, it creates a signal on the *empty?* Flag that we directly feed into the *spawn?* flag of all three cloners. So, one kill leads to three new planes, and *theoretically*, this should spiral out of control quickly. Evidently, it does not. You'd be hard pressed to see more than five Su-25T at the same time. The question is: why?

The explanation is hidden inside the description of the *empty+1* attribute:

empty+1	Number	When all units **from the last clone cycle** have been destroyed, this flag's value is increased by 1

So, it's not that just any cloned group triggers the production of new clones – it must be the last one that the cloner spawned. When a new batch of clones spawns, the cloner forgets about whatever it cloned previously, and only watches the new clones. In our mission, when the first of the three groups is destroyed, all cloners spawn new clones, remember the new and forget about the remaining two groups of 'older' clones that are still alive. If those are subsequently destroyed, no additional clones are produced because their cloners have forgotten about them.

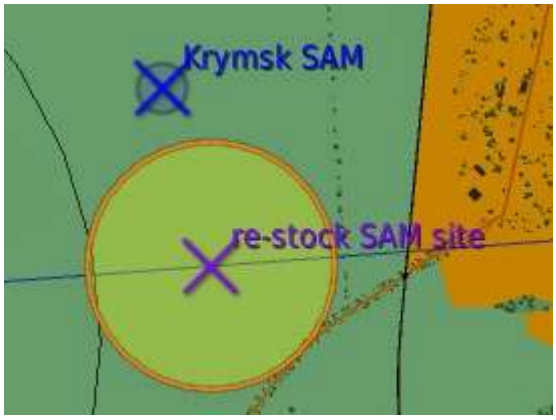
HOW DO THE Su25T FIND THEIR TARGETS?

How is it that the cloned Su-25T that start at different locations all fly to the rallying point and correctly attack the runway instead of missing it by miles?

The answer to this one lies in the fact that when a group is cloned, so is its route. When a clone zone use a foreign template through the *source* attribute, the cloned unit's initial point (the position that it spawns on) may be miles away from that of the original's. Since route points are given in absolute locations, this can lead to unintended consequences, and cloners support the *moveRoute = true* attribute to move each route point by the same offset as the unit's initial point. In our case, however, we want all planes to rally at the first waypoint after their initial point, which is accomplished simply by omitting the *moveRoute* attribute. This way, wherever the cloners are positioned, the cloned aircraft starts inside the clone zone, and then head for the first waypoint which remains the same map location for all cloned planes.

REGULARLY RE-STOCKING SAMS

There is a purple Trigger Zone 're-stock SAM site' south of the Krymsk SAM cloner. It is currently deactivated (paused=yes). If you remove the 'paused' attribute it starts running, emitting a pulse every 5 minutes on flag 50. Flag 50 is the input for Kymsk SAM, causing it to re-spawn. Since it's preWipe attribute is also set, every 5 minutes, the entire SAM site is first deleted, and then replaced with a fresh copy.



While this re-stocking works and is a good way for your missions to provide endless stocks of ammunition for your troops, be mindful of a side-effect here: when we cause the SAM site to respawn, it takes a new random template, so every five minutes, the SAM site not only gets refreshed, it also may get replaced by a completely different configuration. This works as designed, so if you use re-spawn to re-stock a cloner, make sure that you do not also add template randomization to the mix.

5.11 Flag Fun – LOCAL FLAGS (& raiseFlags)

5.11.1 Demonstration Goals

Oh no, more colorful smoke! Well, sit down, and hold on to your shorts, sonny. This one will knock your socks off!

That's because effect that we use for demonstration is rather lame. The underlying technology, however, will make you grin in anticipation of how *you* will use it in your missions.

DML has a number of outstanding features, paramount among them it's ability to use named flags (which is great), and local named flags (which is simply god-like). In DCS, flags are similar to radio channels: they uniquely identify a channel that anyone can tune to. Channels are identified by numbers only, and that is that. DML adds two additional features: channels identified by alphanumeric designations, and 'room only' channels – channels that only those people can receive who are in the same room: zone-local flags. Only modules that are attached to the same zone can transmit on these flags. Even better, other zones can use the same local flag name without getting any crosstalk from other zones.

5.11.2 What To Explore

5.11.2.1 In Mission

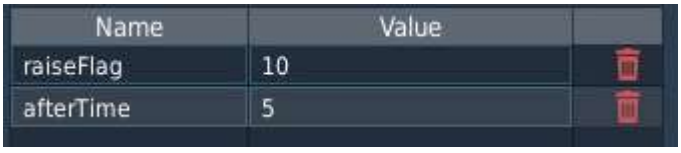
Run the mission. Enter 'Groundhog' at Senaki-Kolkhi. Stifle a yawn. Yeah, colorful smoke starts erupting around the Frogfoot. There's a message after some time. If you run the mission a couple of time, you'll see that the smoke start sequence is different each time, and that the colors for each smoke is also randomized. Nice, but this has been done before.

Now jump into the "Frog From Above" Frogfoot, and kill the units on the derelict airstrip next to the red smoke markers. Note that the northern unit (a BTR-80) keeps respawning after each kill (it sakes a few seconds to respawn, but respawn it will), while the southern unit (a Leo) does not respawn. Again – nice, but we have done this before.

Fire up ME and load the mission. Secure jaw.

5.11.2.2 ME

At first glimpse, there is little surprise: we see a ton of green zones called 'smoker-xx' that we guess contain smoke zones. Off to the side is a zone called 'Fire ME flag'. Let's look at that first:



raiseFlag does exactly what is says on the tin: set a flag (in this case Flag 10) five seconds after the mission starts. Looking at the mission triggers, we see that there is a ONCE trigger "DML called!" that simply puts out a message "DML called and wants its flag back!". The object of this demo is entirely underwhelming yes, you can use raiseFlag to set a ME flag.

Now open one of the green zones, let's examine it and see if we can deduce what it is doing. Click into the "smoker-1" zone and inspect the attributes:

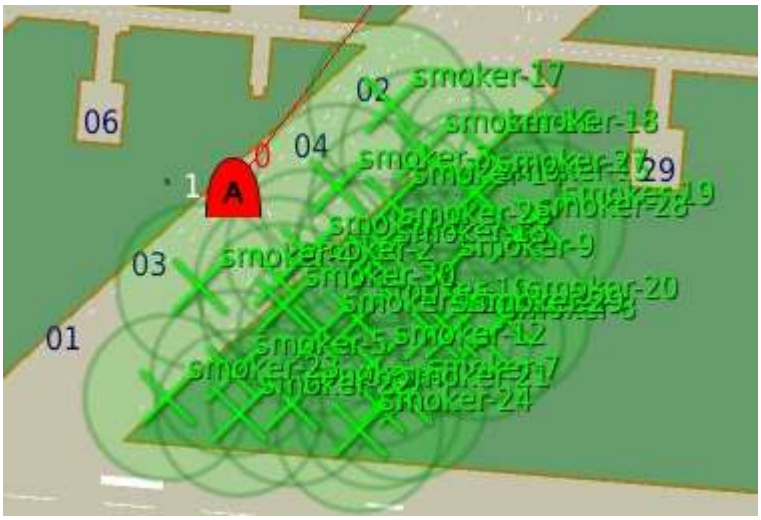
Name	Value
smoke	random
paused	yes
startSmoke?	*smoke ON
raiseFlag	*smoke ON
afterTime	1-20

- Right off the bat, we see that this is a compound zone: it's home to a smoke zone ('smoke' attribute) and raiseFlag module ('raiseFlag')
- The smoke is dormant on start because the paused attribute is present and set to true (yes)
- The smoke zone will trigger when the startSmoke? flag is changed. This flag is set to '*smoke ON' which looks a bit strange (actually, it looks extremely strange to people just coming from ME, as ME only supports numbers as flags. DML supports named flags, so 'smoke ON' is a perfectly legal flag name) but so far, so good.
- The raiseFlag module is set to bang! on flag '*smoke ON' (same strange flag designation), so we know it will eventually start the smoke – simply because the smoke module is wired to the same flag.
- raiseFlag is set to fire 1 to 20 seconds after staring the mission

So what we can deduce from this zone is that will start a randomly colored smoke some 1-20 seconds after the mission starts.

And now for the kicker:

Inspect the other green "smoker" zones. Strangely, they are all alike – the only thing that is different for each zone is it's name. In fact, it looks as the name was assigned by ME during lazy copy/paste. And that that's exactly what happened. How can that be? Why doesn't the first raiseFlag module that bangs on "*smoke ON" start all smoke zones?



Because these flags are all local flags, as indicated by the leading asterisk "*" in their name. Such a flag can only be seen by other modules inside the same zone, and they can't be seen by same-named flags in other zones. So, although the flag has the exact same name in all zones, **"*smoke ON" is a different flag inside each zone**. It can't be seen outside the zone, it does not pollute your mission flag space, and the zone it belongs to *can* be pasted willy nilly all over the map without breaking your mission!

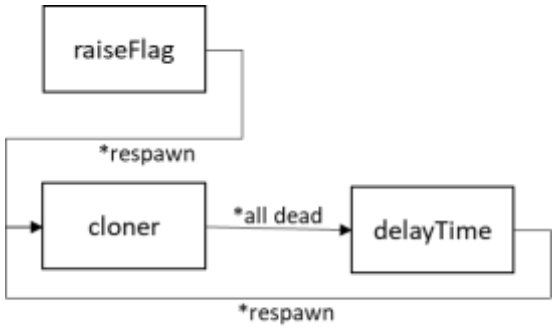
Now recall the respawning BTR-80? Let's check out the spawner to see if we understand how that one works.

Name	Value
raiseFlag!	*respawn
cloner	
clone?	*respawn
empty!	*all dead
timeDelay	2-4
startDelay?	*all dead
delayDone!	*respawn

We see the raiseFlag! Module that, without any other attributes bangs on (local) *respawn – which goes into the cloner's clone? input, causing a clone cycle.

Once all clones are destroyed, this generates an output via empty! on (local) "*allDead", which is wired directly into the delay flag's startDelay? Input. After a delay of (randomized) 2 to 4 seconds, delayDone! Bangs on *respawn, which causes a new clone cycle.

So, we have a basic endlessly respawning, randomized cooldown, clone zone. It uses all local flags. So how difficult is it to adapt this cloner so we can have the Leo at the other end of the runway to also respawn endlessly? Which attributes to we need to adapt?



Let's find out. Copy the zone (ctrl-c) and paste it over the Leo on the other end of the runway.

You are done. Since we are using only local flags, no attributes have to be adapted, the entire 'clone automaton' works autonomously, and **can be copy/pasted anywhere with no change**!

5.11.3 Discussion

Another mission without any Lua in sight!

The cloners use raiseFlag instead of onStart for multiple reasons

- We are about to deprecate onStart for most modulese
- Using raiseFlag makes this spawner more flexible and useful: instead of spawning at mission start, using raiseFlag allows us to introduce an arbitrary start delay for this cloner.

And now for some exercises:

- imagine that you want to make all spawn stop any further spawning when flag 900 is changed. How could you go about that? Hint: the easiest way does not involve the clone.
- Have a message appear per cloner before it respawns, but *not* the very first spawn. Hint: make some 2-4 seconds before the respawn.

5.12 Once, twice, three times a maybe.miz (Event Count Down)

5.12.1 Demonstration Goals: The ~~un~~ending Spawner

This mission demonstrates how we can employ a counter to limit a cloner's spawning to a set number. We do this the following way: we use a flag change to initiate a spawn. Once the determined number of spawns is reached, we no longer want the cloner to spawn.



The set up is easy: we use a cloner that is set up to spawn units at mission start, and then every time all spawned units are destroyed. We can do this with a simple feedback loop on the cloner from *empty!* to *spawn?*. This, by itself, creates an unending spawner as we have seen in *Attack of the CloneZ*.

We now want to limit this in such a way that after the third spawn (including the initial start spawn), the cloner no longer spawns. As additional bonus, we want several smoke zones smoke start their smoke effect to signal that this has happened.

In a normal mission you would use similar configurations to allow something to happen for a number of times before another, different action is initiated.

5.12.2 What To Explore

5.12.2.1 In Mission

Jump into your trusty Su-25T. If you don't know how to use it to destroy ground units, invest the half hour in training, then come back 😊!

Now, in front of you is the target range, and an armed Hummer has just spawned. Switch to A/G, TV and Laser, and use your *Vikhr* missiles to destroy the Hummer. As soon as the Hummer is destroyed, a new one spawns. Circle around and destroy that one as well. Again, a new Hummer spawns. Once you destroy that one, red smoke columns start rising on both sides of the runway, an no more Hummers spawn.

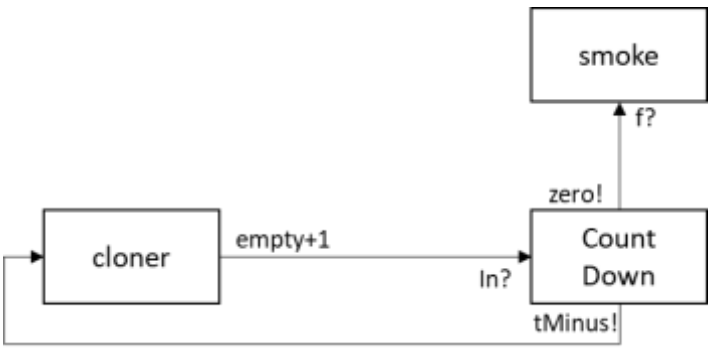
5.12.2.2 ME

There are few surprises here: we have a cloner that spawns the Hummer, and it is set up to spawn *onStart*, and every time there is a change on *spawn?*, which is set to watch flag 100. Every time all units of the last spawn are destroyed, the module bangs flag 100 (*empty+1*) – not 100 as we initially stated as we need a way to interrupt the feedback. If we banged flag 100 with *empty+1*, each kill of the Hummer will spawn a new Hummer, with no way to interrupt it.

Name	Value
cloner	
onStart	yes
spawn?	100
empty+1	110
preWipe	yes

So, to interrupt the feedback loop at the appropriate time, we feed it into a count down module's input to count the *empty+1* events, and use the count down's *tMinus!* output which propagates any input unless the counter has reached zero or less. This completes the circuit from the cloner's output *empty+1* to the cloner's input *spawn?* flag. All that remains now is to set up the count down start value itself, which we set to 3 (note that, according to the documentation, a start of 3 will result in 2 changes on the tMinus! flag, which is exactly what we want).

Name	Value
countDown	3
count?	110
tMinus!	100
zero!	200



Understanding this becomes surprisingly simple if you take a design approach: draw this out on paper, using boxes for spawner, and lines for flags that connect to the in/output attributes. The end result looks similar to a digital circuit, and that is pretty much what we are designing here.

The last bit that completes the picture is quite easy: we connect the smoke zone's *f?* input to the counter's *zero!* output, which causes it to start smoking when the counter's count reaches zero (after three kills).

5.12.3 Discussion

That was easy. And no Lua.

Now, for a small challenge: why don't you build in some cool eye candy: change the set up so that after the Hummer is destroyed, the next Hummer spawns after a 5-10 second pause.

Hint: all you need is a delayFlag.

And now, try to make add some randomization: make the Hummer stop respawn after 3 to 5 re-spawns.

Hint: You don't need anything else, it's all there already 😊

5.13 Bottled Messages.miz (Messenger, timeDelay)

5.13.1 Demonstration Goals

This one is short and sweet, and demonstrates a couple of important “DML glue” concepts: how ME and modules stick together:

- Using Radio Flags to initiate DML actions
- Using multiple modules that “stack” on the same zone
- Using a timeDelay to sequence DML events

5.13.2 What To Explore

5.13.2.1 In Mission

Enter your Frogfoot, and go to Communications→Other and choose “Immediate Radio Check”. This user interaction via Radio menu starts a (very short) sequence of actions from DML: it causes a messenger on the “100 Radio Check” zone to output a message, and play a sound file.

Now choose Communication→Other, and select “Radio Check, delayed response”. Another, slightly longer sequence of DML actions runs:

- A message “Tower, Frogger One, radio check” appears
- After a short while, “Read you five” appears on screen, and a sound file plays.



That’s all, folks!

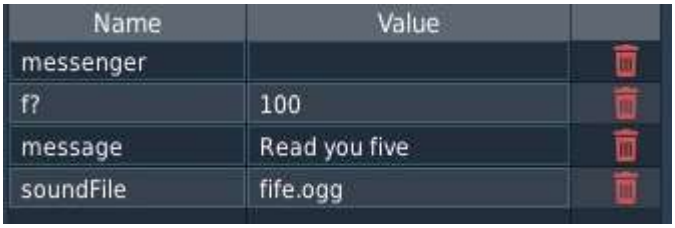
5.13.2.2 ME

In order to create user events, we use ME’s “Radio Item Add” actions that create a new entry in the Communications→Other tree.



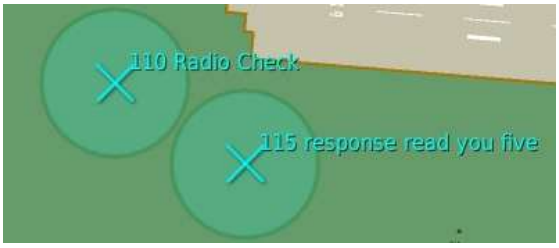
These radio items, when chosen, cause a flag (here 100 and 110) to be set to a specific value (note that ME can’t increase flags with radio items, just set them to a specific value, so any subsequent choice of the same radio item will not register as a flag change).

DML modules detect this change to initiate their actions.



In a simple (the first) case, it simply activates a messenger (*f? 100*) which causes it to output a *message* ‘five by five’ and plays the audio *soundFile* *fife.ogg* (note that this sound file is added to the mission by a separate trigger to ensure it is included into the .miz archive).

A more interesting cascade is started when we choose the “Radio Check, delayed response” option: We first see a message “Tower, Frogger One, radio check.” displayed on our screen (mimicking us requesting a radio check from the tower), followed by a brief (variable) pause, and then the written and aural response “read you five”). The most interesting bit here is the delay. This allows us to insert a pause between DML actions that seems more life like (in this case). In general, though it’s often desirable to allow a brief time interval between actions, and timeDelay is the perfect tool to do this for you.



This is achieved by multiple chained DML modules: The initial 110 flag change from the radio item activates a combined timeDelay/messenger zone: it activates *f? 110* which is common to both messenger and time delay. The message is displayed (messenger) and a delay is started (*timeDelay 1-2* seconds, randomized). When the time delay is complete, *timeDelay* bangs *out! 115*, which is (not coincidentally at all) the input flag for the second messenger.

Name	Value
messenger	
message	Tower, Frogger One, radio ch
f?	110
timeDelay	1-2
out!	115

The second messenger, attached to zone “115 response read you five” offers little surprise and is configured nearly identically to the original one-step action. Once the message and audio are played, this cascade ends.

Name	Value
messenger	
message	Read you five
f?	115
soundFile	fife.ogg

5.13.3 Discussion

COMPOUND MODULE USE

Using compound modules can be both a blessing and a curse. Most commonly the following modules are used together with others:

- delayTimer (to chain DML actions)
- messenger (to annotate actions)

so it’s fitting that we use these two together to demonstrate the concept. When you use modules together, there are several points that you need to keep in the back of your head to avoid unintended consequences:

SHARED ATTRIBUTES

When two modules per their “ME Attributes” description have one or more attributes with the same name (e.g. *in?* for messenger in Clone Zone) you must ensure that this is in line with your intended use. Usually, they are for trigger handling (inputs always end on a question mark, and outputs should always end on +1 or an exclamation point). Some modules may share a name, or use similarly named attributes that can lead to confusion.

Also, mind that it's usually not a problem if two or more modules share the same input attribute (e.g. in?), it *can* be problematic if two or more modules share the same output attribute, and fire at the same time. In that case, the way that modules change the output flag can come into conflict.

TIMING, INPUTS AND OUTPUTS

Remember that even though modules may "share" the same input and output attributes, internally they may be treated quite differently. All modules run on their own 'clock' (meaning they use different times when they look at the input flag or bang an output flag), and it's virtually guaranteed that these clocks are *not* synchronized. Therefore, you never know which module reads a flag first, nor how much time passes before other modules check that flag. Don't ever rely on an order in which modules react, and always ensure that there are no dependencies on the order in which they do.

RICH MAN'S DEBUG TOOL

When using Lua, `trigger.action.outText()` is pretty much functions as DCS's poor man's debug tool. DML's messenger module is the rich man's equivalent. Attach it to a zone that you want to monitor, queue it to the same trigger, and have it report whenever the module that you are interested in activates. When done, simply remove the messenger attribute.

5.14 Follow Me! (unit zones & messenger)

5.14.1 Demonstration Goals

This mission demonstrates how unitZones can be used to detect units / players and how that information can drive messengers to provide feedback to create mock ground controller or formation flying trainer.



5.14.2 What To Explore

5.14.2.1 In Mission

Hop into the Frogfoot on the ground. Once inside, increase throttle to start rolling, and close the distance to the Hummer in front of you. It starts rolling, and depending on how close you are will encourage you to get closer, or remain at that distance.

Once you reach the runway threshold, you are told that you should contact tower for clearance, and the hummer veers off to the right, while you should stop. You no longer receive messages that you should get closer to the Hummer.

Now restart the mission and jump into the airborne Frogfoot. In front of you is another Frogfoot ‘Maximo Lider’. Find out how good your flying skills are, and fly in close formation (less that 30m apart). When you get inside 60 and 30 meter distance, you’ll get messages, and when you then widen the gap, you’ll again get messages when you grow more than 30 or 60 meters apart.

5.14.2.2 ME

Let's begin with the ground-bound Frogfoot. We have a zone "Detect Player One" that ostensible exists to detect the presence of a player plane (and serve as a set-up for a cheap pun later). When we inspect the zone itself we find that it is a compound zone, consisting of a unitZone and a messenger module.



The unitZone looks for player-controlled planes (*matching* is set to *player*) whose name begins with "The"(lookfor = "The*" – note the asterisk at the end to denote wildcard). Note that both client (player) planes match that description, for simple expediency (so I could re-use the zone via copy / paste).

The zone is also set up to bang! the local flag 'hi there' when the player unit is found to enter the zone. When all player planes that match the description leave this zone, the module is also to bang! on (ME compatible, global) flag 100.

Name	Value
unitZone	
lookFor	The*
matching	player
enterZone!	*hi There
exitZone!	100
messenger	
messageOut?	*hi There
message	Frog One, approach and foll

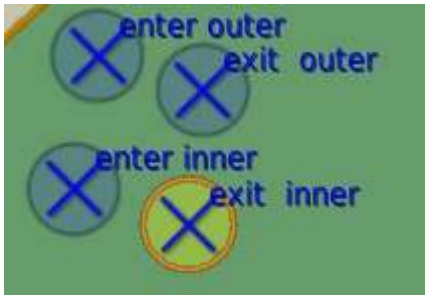
Stacked on the zone is also a messenger flag that activates when local flah 'hi there' changes, and that displays the greeting message that you saw when you entered the cockpit on the ground.

So what is that ME flag 100 for? Well, that is an old-school shout-out to ME that allows the Hummer to proceed with its path to waypoint one (it was held at the initial waypoint by a waypoint action. Accordingly, we expect the Hummer to start rolling when we move out of the "Detect Player One" zone.

So how did the Hummer keep telling you that you should come and that you are close enough? This is accomplished by the two essentially identical, concentric unitZones "inner" and "outer" that move centered on the Hummer (linkedUnit = FollowMe) and that stack a unitZone to generate flag change events whenever the player moves inside or outside of the zone. Each entry and exit of the zone generate their own global DML events, a total of four (4) different events.

Name	Value
linkedUnit	FollowMe
unitZone	
lookFor	The*
matching	player
enterZone!	enter outer
exitZone!	exit outer

And that is solves the secret: there are four separate ME Trigger Zones that each carry a messenger. Each messenger is tied to their own trigger event, and thus create the four different messages that you encounter:



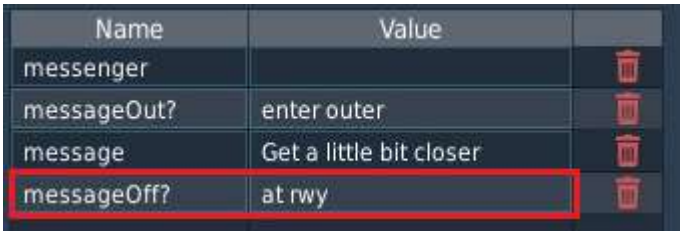
- "Get a little bit closer" when you enter the outer zone
- "That's close enough!" when you enter the inner zone

- “You could get a bit closer” when you exit the inner again and
- “Frog One, keep up!” when you leave the outer zone.

There is one last messenger feature that we use to “mute” all messages when the Hummer reaches the runway. At the runways hold short position we find a quad trigger zone “Ready Player One” that also contains a unitZone module. This unitZone looks for the Hummer’s Group (lookFor = Follo*, and no ‘matching’, which defaults to Groups). As soon as the first (and only) unit of that group enters the zone, it bangs! on the (global DML) flag “at rwy”.



The four messengers that we looked at before all have this flag wired to their “messageOff?” input. Once a signal is received on that input, a messenger goes silent, so all four messengers immediately stop sending out messages when you enter/leave the zones around the Hummer – to the player it looks as if control was handed over to Tower.



Now look at the set up for the airborne version, and you’ll see that it’s an almost identical setup. This is only provided to show a couple of things:

- unitZones update quickly enough to also allow inflight following (they use just-in-time updates, so they only update once a second, every time they are required to check so have negligible performance cost)
- You can use simple combinations like this to create complex flight following/flight training missions where players must stay close to a unit and receive warnings when they are getting too far away
- Just how cool are messengers that you can tell to shut up?

5.14.3 Discussion

Yeah, no Lua.

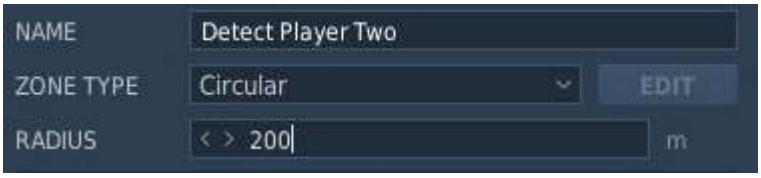
Detect Player One

This demo uses a pretty underhanded trick to detect when the player enters the ground-based Frogfoot that only works under certain conditions:

- unitZones are initialized at start-up and save if initially, the units they look for are inside or outside the zone so they can respond correctly when later on the units enter or leave the zone. When *at start-up* it is determined that a looked-for unit is inside the unitZone, no signal is generated, but the state is saved.
- So how is it possible that out ‘Detect Player One’ unit detects the player when we enter the mission at the beginning? This is due to two factors:
    - unitZone must load at startup (as done in the ONSTART trigger)

    - A peculiarity of missions is that when the mission starts up, player units are not yet inside the world – players look at the slot choice panel.
- When the players choose their planes, unitZone has initialized, and saved the fact that no player plane is inside “Detect Player One”. So when it suddenly detects a spawned player unit inside, the zone creates a zoneEntered! flag change.

Detect Player Two



Why is this zone so big? Because the player plane that spawns is fast, and it may exit the zone during the interval that unitZones samples all zones (by default once per second). In ME, we set The Flying Frog up to start at 500 km/h, which translates to some 140 m/s. We therefore add some margin for safety, and make the zone's radius 200m (400m across), so even two-seconds between samples will allow the zone to detect the player plane when it spawns.

Now, there is an easy fix that would allow you to use a small zone instead, irrespective of the player unit’s speed. Can you find it?

Spoiler: yes, it involves a linkedUnit attribute.

Detect the other player?

Both Detect Player zones use a lookFor = “The*” wildcard that fits both player planes: “The Frog” and “The Flying Frog”. So what do you predict would happen if you fly “The Flying Frog” through “Detect Player One” – and why?

Hint: revert back to the previous topic “Detect Player Two”.

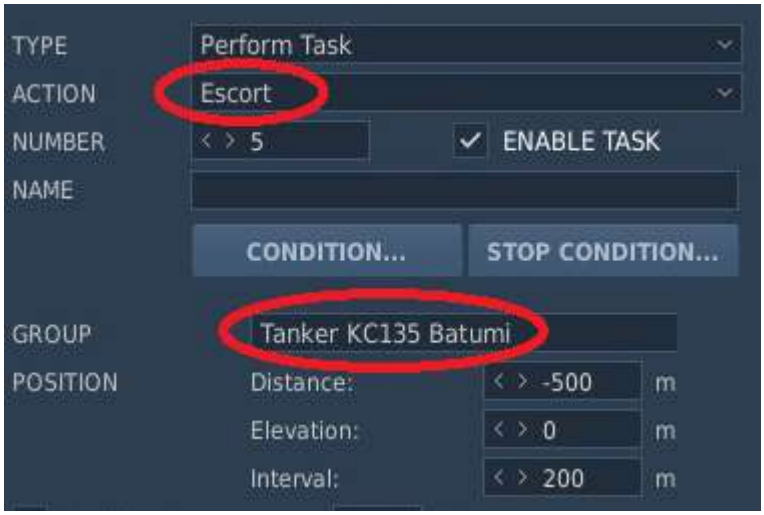
Quads, too!

That zone on the ground, at the hold short position. The one that detects when the Hummer is inside and tells the pilot to contact tower? Yeah, that’s a quad zone. Glad you noticed. All zone based modules support quad zones – as long as you remember to keep the angles below 180 degrees (which usually is the case with most quad zones.

5.15 Clone Relations (Advanced Topic)

5.15.1 Demonstration Goals

This demo explores the question: what happens to units/groups that are cloned and that have other units/groups in their waypoints?



In other words, what happens when either (or all) of them are clones? What happens when

- A unit with such a reference is cloned (i.e. the escorting unit is cloned, the unit to escort isn't)
- A unit with such a reference is cloned and the targeted group is also a cloned

A common example is a rescue helicopter that is placed off the side of a carrier and simply shadows the movement of the carrier through a 'follow group' task. After a while, the rescue chopper runs out of gas and has to land. This is fine for most scenarios, but some servers run 24/7, and here the mission designer runs into an issue: the helicopter disappears after a few hours, requiring some advanced mission design Lua-fu to have it re-appear on station. With a cloner this can be easily remedied, if the rescue helicopters spawn regularly (say every 2 hours), and the clone can correctly resolve it's reference to the unit/group it should follow.



Even more direct are battle groups that engage each other and that should perpetually respawn after being destroyed to simulate an ongoing battle. Here the enemy groups have each other as attack targets. The cloner must ensure that when new clones spawn they attack the correct group, as the one their task refers to was destroyed hours ago, and the group they should attack are clones.

5.15.2 What To Explore

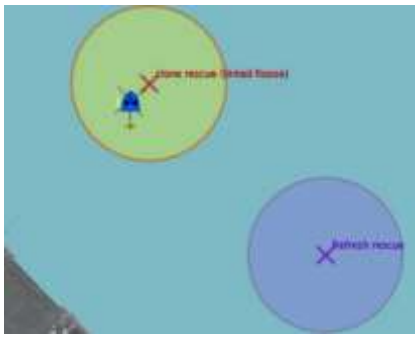
5.15.2.1 In Mission
This mission is a collection of micro-case-studies, and we want to explore them one at a time. Start the mission, then

1. Hit F9 to cycle through the ships until you are viewing the Roosevelt. Notice the rescue helicopter off it's starboard side. Observe it closely. Every 10 seconds it seems to jump slightly, and then resume it's location.
2. Also note the Hornets that take off the Roosevelt every 90 seconds.
3. Now, cycle with F9 until you are looking at the Seawise Giant. Note the *two* helicopters following it. If it's still early in the mission, you'll notice the lagging helicopter closing the gap to the leading, and then keeping it's distance after a short while.
4. Now cycle to the Stennis. Observe the Hornet taking off (you may need to restart the mission), and the deck equipment.
5. Using F2, cycle to the KC135 tanker, and note its escort of three vipers.
6. F2-cycle to the lone chopper flying over water to apparently nowhere
7. More F2: cycle to the Frogfoot, and watch it attack a static target at Batumi (if it survives the encounter with the eagle)
8. Use F7 to cycle to one of the two battling tanks. Notice how they perpetually respawn and attack each other.

5.15.2.2 ME
So let's go through the things we observed in-mission and note what is remarkable about it, why, and how we did it with DML

1. This ("Clone Rescue") is an endlessly respawning rescue helicopter that always appears in the same spot relative to the carrier and then assumes follow position. This is a possible solution for 24/7 servers who want permanent rescue helicopters on-station with a minimum of fuss. It's realized with a zone that has as linkedUnit the Roosevelt (so it moves with the ship) and keeps its relative position to the Roosevelt through the useOffset Attribute. This linked zone also contains the cloner that clones the UH-60. Since preWipe is true, the last spawned instance of the helicopter is despawned before the new one is cloned. The UH 60 has orders to follow the Roosevelt, and when cloned, these follow orders are cloned as well.
   Finally, there is pulser in the Zone (Refresh Rescue) that bangs the cloner's input every 10 seconds to initiate a new clone cycle.

Name	Value
linkedUnit	Roosevelt
useOffset	yes
cloner	
preWipe	yes
spawn?	100



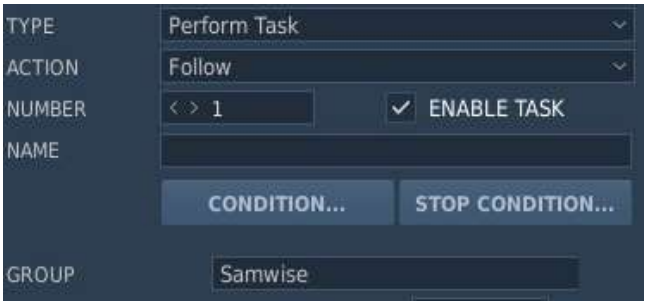
2. The Hornet is regularly spawned by the "Bug Cloner" clone zone. Note that since the Hornet is ser to "Takeoff from Runway" in its initial waypoint, we do not have to set the cloner to follow the Roosevelt via 'linkedUnit' (although it would not hurt to do so). Upon spawn, the waypoint reference

is resolved to the current position of the Roosevelt, and hence the Hormet clone spawns correctly, and can take off. The spawner's preWipe attribute is given to not overwhelm the mission (else a hornet is cloned every 90 seconds, dragging down DCS's performance). Spawning is controlled with the pulser in "Fresh Bugs"

3. There is an important difference between the Seawise and Roosevelt that becomes apparent when you look at them in ME: in-mission, the Seawise is a clone, while the Roosevelt is the original unit that was assigned in ME. This demonstrates two important concepts that DML solves under the hood. Both UH-60 that are following the Seawise have "Follow" instructions (as a Tolkien fan, the Seawise will also be called "Samwise", I can't help it. Ah, "followship" – huh. Puns "R" us.). When the mission starts up, first the "followship of clones" group is cloned. During the cloning process, the cloner recognizes that the unit that the helicopter is to follow is also part of the cloning template, and automatically resolves the following orders to follow the cloned seawise.



A little bit later the "clone following clone" zone is triggered. That group has orders to follow the "Samwise" that was placed in ME, but that unit no longer exists (it was removed during start-up). The cloner then looks through its records and sees that it did clone that unit, and automatically assigned the last clone from the original as target of the follow task. So it is possible (and common) to clone units that interact with other units that are themselves clones. The rule to remember here is that unless the reference is to a unit within the same clone template, a reference to a cloned unit is always a reference to the last cloned instance. Keep this in mind when designing your mission, and it will usually work out well.



4. The (non-SC) Stennis works similar to the Roosevelt. This sample shows that Aircraft can clone onto starting positions without issues. Additionally, we are cloning static objects here onto the Stennis' deck, simply to show that it's possible. This is only important if you want to clone cargo, as you would usually simply use standard (un-cloned) static objects to place on a carrier's deck. There's very little to see here except proof that a cloner can correctly handle static objects linked to moving ships.

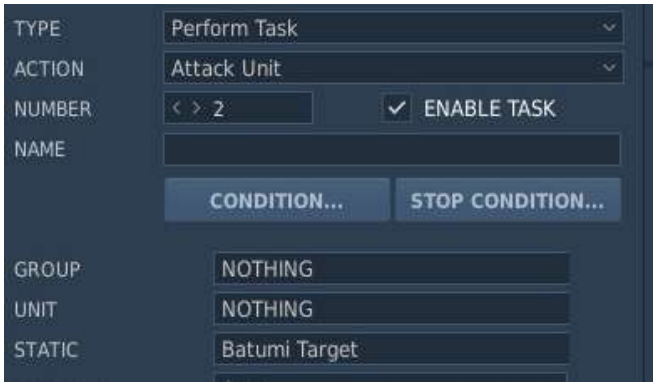


5. This example simply shows how clones (the falcons) can escort a normal unit, it's the purely aerial equivalent of the helicopter following the Roosevelt; it's also proof that cloners support the 'Escort' task in addition to 'Follow'.



6. This little example demonstrates that you can use foreign templates in a cloner and the cloned unit will still faithfully execute their orders.
Although it doesn't look like it, the UH-60 is following the Roosevelt. It merely has to cross the 46 km distance first.

7. This is more interesting, and here we see the application of the 'last clone is it' rule. The Frogfoot has orders to attack the static object 'Batumi Target.
This static, however, is part of a clone template that is cloned four times before the Frogfoot spawns. Consequently, when the Frogfoot is cloned, clone zone updates the Task to attack the static clone that was cloned last (it's undetermined which of the five cloners will be the last one since they all fire during the same cycle. If you need to tightly control which clone to attack, you will have to sequence the cloning process with timers like delayFlag)



8. And finally, the tale of the endlessly battling tanks. Both groups have each other as targets defined, both are endless respawners (meaning their spawn is short-circuited with their empty+1 signal). The interesting question here is: how will this resolve, as theoretically, one will not have a correct target at their initial spawn. While this is correct (whoever spawns first will not have a correct attack target as the other unit doesn't exist), this quickly is resolved by the fact that neither tank survives for for long, and all subsequent spawns always resolve correctly to their last-cloned enemy.

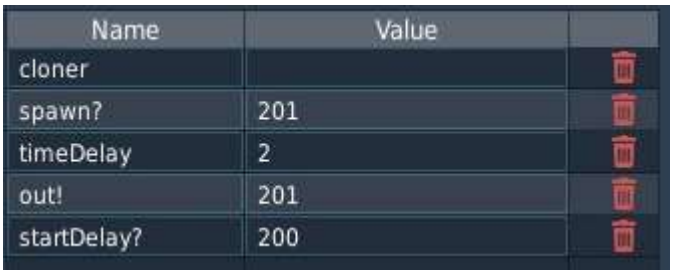


Now, before you get too excited – even though dynamically updated clone tasks is an impressive feat coding-wise, it doesn't do much in this example, as the tanks would battle each other even without having the task to attack each other.

5.15.3 Discussion

Compound Zone Modules / Synonym use
There is an interesting Zone setup that involves the Seawise Giant cloning example. Inspect the Zone “Clone Following Clone”.

What we have here is a cloner that is cloning after a delay, and the delay is built into the zone. The signal enters on flag 200 (which is used to clone the Samwise and the first UH-60), and goes into the dealyFlag module before it exits the delayFlag on 201 and enters the cloner.



This is required to prevent the two clone Zones “Followship of clones” and “Clone Following Clone” to spawn uncontrollably at the same time, the delayFlag module enforces a separation of the cloning process by time (here 2 seconds).

Both modules (delayFlag and cloner) share the same zone - we could have used two separate zones, one each for the cloner and one for the delayFlag module. However, to do so we must be careful to separate the inputs so they would not get crossed. timeDelay has three synonyms for their input flags: in?, f?, and startDelay? – while a cloner responds to in?, f?, spawn? and clone?. In order not net cross out input lines, in? and f? are out, while we are free to use the other synonyms to keep the lines separate. In our example we chose ‘spawn?’ for the cloner, and startDelay for delayFlag.

No carrier clones
Currently, you can’t mix cloned carriers with cloned flights from that carrier. For reasons unexplained, it simply doesn’t work. Place a cloner over the Stennis, and have it spawn onStart. Try as you might, you will not be able to get the hornet spawn from the cloned Stennis. A bit annoying, but since carriers aren’t supposed to die, much less respawn in a mission, this is an annoyance more than anything else.

Perhaps this issue can be resolved in the future.

5.16 Moving Spawners.miz (SpawnZone and linkedUnit)

5.16.1 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



5.16.2 What To Explore

5.16.2.1 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.

5.16.2.2 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** interations. 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

5.16.3 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.

5.17 Helo Trooper.miz

5.17.1 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.

5.17.2 What To Explore

5.17.2.1 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 an 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

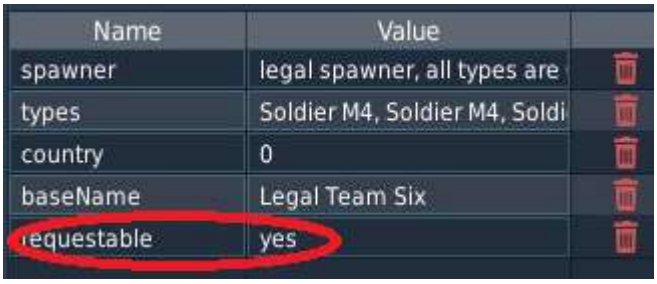
happens the moment that your helicopter lands so you can safely change options while on the ground.

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.

5.17.2.2 ME
Note that this mission shows a couple of important features:



- A spawner's requestable attribute controls if the troops can be requested via a helicopter's menu. Note that simply making a spawner requestable does not ensure that the spawned troops can be picked up. That check is made separately, the troops merely spawn (which can be used for different purposes altogether: a spawn can be triggered if your helicopter is close enough, and this alone can be a mission goal)
- 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.

5.17.3 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 fee 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!

5.18 Helo Cargo.miz – cargo spawn & receive

5.18.1 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.

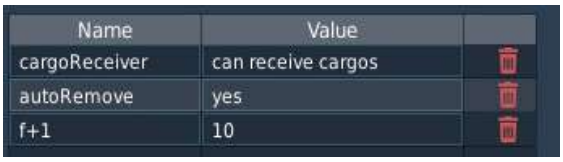
5.18.2 What To Explore

5.18.2.1 In Mission

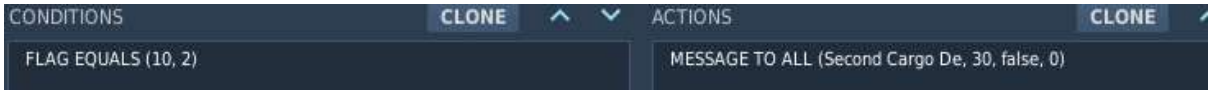
Fly any of the provided helicopters. Use the standard communication menu to pick up cargo. Once hooked, slowly fly the cargo towards the receiver zone that is marked with tires and green smoke. Note that during approach text messages guide you towards the zone. Unhook the cargo inside to deliver. A message 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

5.18.2.2 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**



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.

5.18.3 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, in 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

5.19 Artillery with UI.miz

5.19.1 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.

5.19.2 What To Explore

5.19.2.1 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



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.



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.

5.19.2.2 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

5.19.3 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).

5.20 Missile Evasion (Guardian Angel).miz

5.20.1 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)

5.20.2 What To Explore

5.20.2.1 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 no 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

5.20.2.2 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.

5.20.3 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. Zurn 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.

5.21 Recon Mode.miz

5.21.1 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.

5.21.2 What To Explore

5.21.2.1 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.

5.21.2.2 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 blcklist (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

5.21.3 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.

5.22 Owned Zones ME Integration.miz

5.22.1 Demonstration Goals

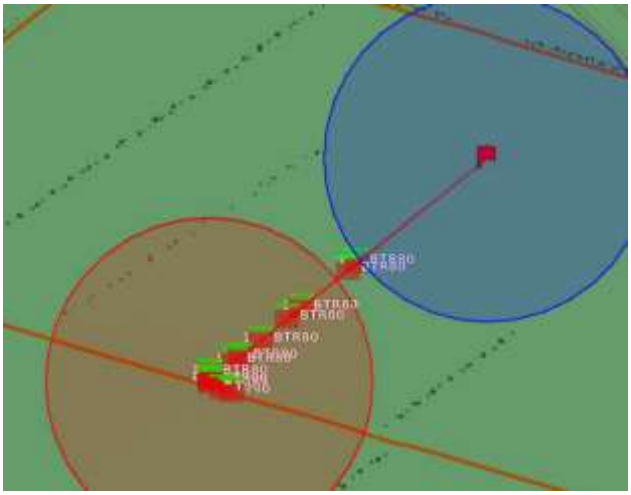
This mission shows how Owned Zones work in general and how they can be used to set ME Flags. It also offers a nice test bed to illustrate how the various cooldowns work and can influence the game.

5.22.2 What To Explore

5.22.2.1 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

5.22.2.2 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 vaue 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.

5.22.3 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.

5.23 FARP and away.miz (tbc)

5.23.1 Demonstration Goals

5.23.2 What To Explore

5.23.2.1 In Mission

5.23.2.2 ME

5.23.3 Discussion

5.24 Keeping Score.miz (PlayerScore)

5.24.1 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.

5.24.2 What To Explore

5.24.2.1 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.

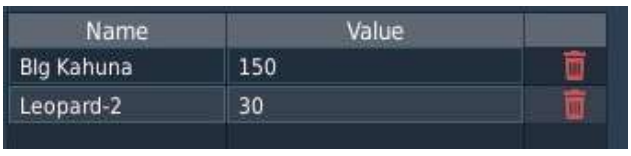
After successfully killing some vehicles (at least one BTR-80 or Leopard, and one T-90), choose Communication→Other… →Score/Kills. This is the Player Score UI module that allows a player access to more detailed personal score information. You are presented with your personal kill log:

- Total score an number of kills
- List of all types killed and their number
- Note that named kills also appear as a separate type



5.24.2.2 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.



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.

5.24.3 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.

5.25 The Zonal Countdown (Local/Global Flag Demo)

5.25.1 Demonstration Goals

This mission demonstrates how you can use a messenger module as a real-time counter, using message wildcards, and how to use a count-down module for the surprising task of implementing a real-time count-down. It also serves as another nice demonstration of DML flags in action, as this module stack is activated by a global ME flag, and then runs on internal flags until it triggers the explosion which is done in ME



We got an extra prop from the set of “Speed” for this demo!

5.25.2 What To Explore

5.25.2.1 In Mission

Jump into your trusty Frogfoot. Follow the instructions: trigger the countdown with Communications→Other→Bye Bye Bus.

A countdown appears on-screen. Note the following:



The text claims to be a message from zone “Countdown”. And it shows a continually changing number that rapidly approaches zero.

Unsurprisingly, when the number reaches 0, the bus goes bye, bye in a rather spectacular way.



5.25.2.2 ME

To start the count-down, we see a standard ME trigger set up on mission start with a radio item which sets flag 100 when selected.

ACTION:	RADIO ITEM ADD
NAME:	Bye Bye Bus
FLAG:	< > 100
VALUE:	< > 1

This gets the ball rolling, and is a standard UI pattern for player-induced actions.

On the map itself we see a couple of units, and two trigger zones. The units are no surprise. The “Countdown” zone is the ‘automaton’ zone, a little stack of modules that work together to perform the countdown, provide timing, and provide the real-time messaging. The “Zero-Message” zone is only present to provide additional visual candy (it provides the final ‘Bye, bye Bus!’ message to remove the count-down message.



The “real” work happens in the “Countdown” zone, and, we’ll take it apart one by one:

We begin the stack with a **pulser**. This is merely a ‘clock’ that provides a regular signal on the local flag *thePulse. At the start of the mission, the pulser is stopped (pulseStopped = yes), and once the (global) flag 100 changes, the pulser will start sending out a signal on *thePulse once every second (time = 1)

pulse!	*thePulse
time	1
pulseStopped	yes
startPulse?	100

Once the pulse starts beating, it feeds into a **count down** module. This module starts counting at 11 (‘this one goes to eleven!’), and every time a signal is received on local *pulse, it counts down on. The current value of the counter is passed out on local *cVal for anyone who is interested (the messenger is). Also, every time the value changes, tMinus! bangs out on local *counted (which is, strictly speaking, not necessary and done here for clarity; *cVal could also have served the same purpose to drive messenger’s messageOut). When the count down reaches 0, it bangs on the global (and ME-compatible since it is a number) flag 110, and global named flag “boom”.

countDown	11
count?	*thePulse
counterOut!	*cVal
tMinus!	*counted
zero!	110, boom

The **messenger** module is triggered every time a signal arrives (the flag’s value changes) on (local) *counted. The module then reads the current value on (local) flag *cVal (which happens to be the value of the counter). The message that this module is instructed to put on the screen is “*This is a text from Zone <z>. Countdown: <v>”*. The two wildcards <z> and

message	This is a text from Zone <z>
messageValue?	*cVal
messageOut?	*counted
clearScreen	yes

<*v*> are replaced with the zone's name ("Countdown") and value of the flag connected to messageValue? (*cVal).

Before it's put on the screen, the screen is cleared, removing all previously displayed messages, and moving the output line to the top.



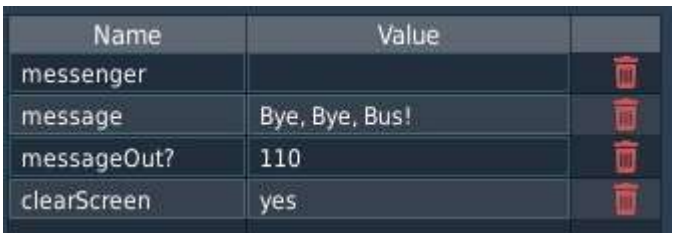


This is what makes the countdown seem to be updated; in reality, it's merely overwritten.

When the count down reaches zero, it bangs on (global and ME-compatible number) flag 110. This is detected by the Mission Trigger "ONCE, Boom", which simply explodes the Bus when the flag 110 changes to a non-zero value



At the same time as ME is busy blowing up the bus, the messenger in the "Zero Message" zone gets active: triggered by the same signal on (ME compatible global number) flag 110, it clears the screen (erasing the last count down message), and writes Bye, Bye, Bus!" to the screen. This is a purely cosmetic addition and not necessarily required. It's also a bit wasteful, as we need another zone (two messenger modules don't stack on the same zone). There's even a third zone ("Second Zero") which we'll look at in the Discussion section. But someone told me that it usually pays to be thorough and dot your Tees and cross your eyes 😊.

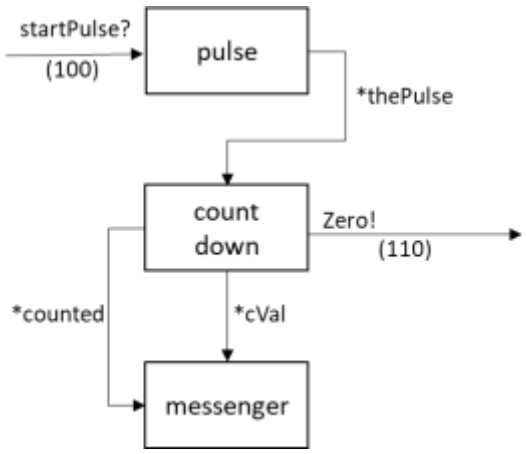


5.25.3 Discussion

Another mission, another no-show for Lua.

DESIGN DETAILS

Just a minor observation: since the messenger reacts to changes on the messageOut? Input, we could have fed *cVal instead of *counted into messageOut? And received the same results, saving one local flag.



But since DML flags add very little cost, and make things much clearer, we used a separate flag to tell messenger that a new message should be displayed.

Again, I strongly recommend that you design your zones with block diagrams (as shown oh so beautifully on the right) either while or before you create the mission, as it makes it much easier to spot potential issues, and makes choosing local flag names easier. It also helps to identify global flags early in the process.

THE RACE: SECOND ZERO
Also of interest is the second flag that is set when the counter counts down to zero: It sets two global flags: 110 (number-name) and “boom”.

Both of these flags trigger a messenger, each one in a different zone (110 in “Zero Message”), and “boom” (in zone “Second Zero”). When you use multiple outputs like we do here, you must be observant of a peculiar effect that can arise in programming: a race condition. This is when two or more processes are running concurrently, and the order in which they complete is important.

When the counter’s zero! output fires, both flags 110 and “boom” are simultaneously banged. There is no way to determine which one will be picked first, so you don’t know which one will write to the screen first. If you added the *clearScreen* attribute to one of the two messengers, in roughly half the cases they’d be processed after the other messenger, and that messenger’s message would be erased. So none of them must have a *clearScreen*, to ensure that this never happens; the order in which you define the Zones or list the flags in counter’s “zero!” output has no impact on which of the two will be processed first. Keep this in mind when you design missions: when you rely on the assumption that one thing happens after the other, you may have to ensure that they do. A delay flag is one method of ensuring a sequence. xFlags is another.

CHALLENGE:
Try to build something into “Zero Message” and “Second Zero” that ensures Second Zero is always processed after Zero Message” and that allows you to add the *clearScreen* attribute to Zero Message.

5.26 Frog Men Training.miz

5.26.1 Demonstration Goals

This is a fully developed mission to facilitate Su-25T weapons training. It demonstrates the use of multiple modules that work in conjunction with (very few) traditional ME triggers to achieve various purposes:

- Unit/Group activation/spawning when certain groups spawn
- Endlessly replenishing enemies
- Protection from Missiles
- ‘Restocking’ SAMs after some time
- ‘Repairing’ suppressed SAM sites
- Messaging when certain spawners activate
- Using compound/stacked module activation on zones



5.26.2 What To Explore

5.26.2.1 In Mission

There are multiple Su-25T configurations to choose from:

- SEAD
- CCIP/CCRP bombardment
- Laser Guided Missiles
- TV Guided Missiles
- Gun Pods
- Air to Air

There is a big main target range on the old airfield SW of Kobuleti where most of your target practice occurs. The targets are set to automatically respawn once a group is fully destroyed.

You can change air frames at any time. When choosing some air frames, observe the following

- When choosing Air to Air (*Air Frog One*), a group of A-10 and a group of Ka-50 spawn as targets in front of you
- When choosing Gun Pods (*Convoy Gone*), a convoy spawns in front of you close to the orange smoke
- When choosing *SAM Hunter*, a couple SAM sites spawn and come on-line a few seconds later. Since you are protected by Guardian Angel, the SAM sites should not be able to kill you. When you completely kill a SAM site, it respawns after the last unit is destroyed. You have some 30 seconds before they start re-acquiring you.
  Also observe that after some time (without attacking their radars), the SAMs stop firing at you as they have run out of missiles. After a while, though, they will start again – they have been replenished.
  Finally, observe that if you have suppressed a SAM site by destroying their radar, some time later they come back on-line and attack you

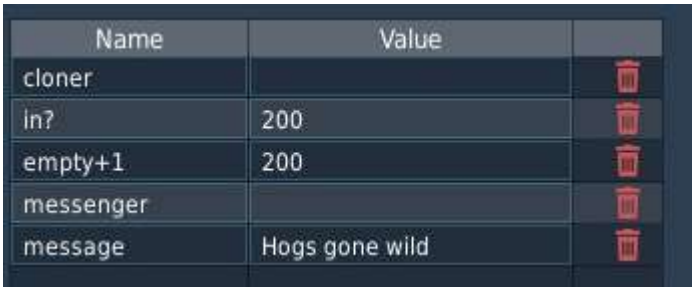
Above groups will not spawn unless you enter the appropriate airframe. Once they have spawned, however, they will remain in-game (remember this when you spawn the SAMs) even if you change to a different configuration

Destroy entire groups on the ground to see that they automatically re-spawn. The same is true for SAM sites, Helicopters, and Aircraft.



5.26.2.2 ME
This mission assembles lessons from previous demos and add some more advanced topics. It is a fully functioning mission that's also available for download.

COMPOUND ZONES
In this mission, we stack multiple modules onto the same zone. This serves to make access easier, as well as grouping logically connected functions into the same visual representation. Look for the zone.called "Hog cloner", far up in the north, North of Maykop-Khanskaya.



Inspecting the attributes, we see that there are two module keywords: *cloner*, indicating that cloneZones attaches to this zone, and *messenger*, meaning that also that the messenger module anchors to it. If you look at both zones' descriptions, you will find that they share one attribute: *in?* (200) – the attribute that controls which flag to watch for a change to occur. Since both modules share the same attribute, they will also trigger with the same flag

change (technically, it's a bit more involved, but that's what it boils down to). So, every time flag 200 changes its value, *both* cloner and messenger activate. Note that it is not determined which activates first, there is no inherent order in which modules are triggered.

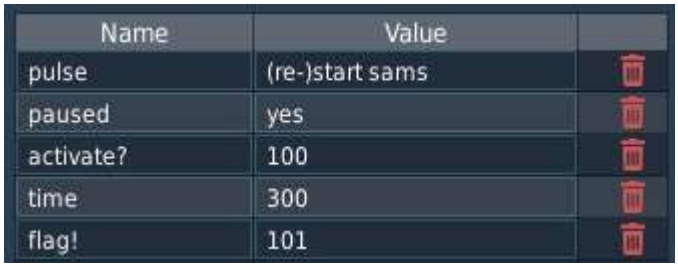
The upshot is that every time the cloner spawns a new set of clones, the messenger also plays a message.

ENDLESS SPAWNS

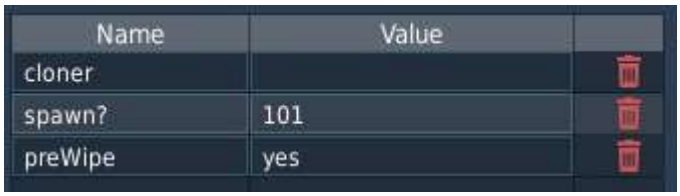
This is achieved by creating a feedback loop from *empty+1* into the *in?/spawn?* input of the cloner. Whenever the last unit of the previous clone is destroyed, *empty+1* fires, which feeds into *in?/spawn* creating a new clone cycle.

Note that the SAM cloners do *not* use a feedback loop. Read below why that is not required.

RESUPPLYING/REPAIRING SAMS



In order to re-supply and repair the SAMs, we use a simple idea: simply remove (*preWipe yes*) the existing group, and them immediately replace it with a fresh copy of the (fully stocked and fully repaired) template. We do this regularly, say every 5 minutes (300 seconds). For this we can use a pulser set to change a flag (say 101) that causes a spawn at the spawner every 300 seconds. This also neatly solves our endless respawn requirement. If by some miracle you are fast enough to completely wipe a SAM site within the five minutes after it spawns, it will still respawn after 5 minutes and we do not need a feedback loop from *empty+1* → *spawn?* for the SAM spawners.



Configured this way, this pulser, when running, causes flag 101 to change every 300 seconds. Since 101 is connected to the spawn? input of the SAM cloners, they regularly cause a re-spawn of all SAM units every 5 minutes (5 * 60 = 300 seconds). Since the SAM spawners are set to *preWipe yes*, any remaining units from the previous spawn are removed from the game when a new spawn cycle is started, resulting in fresh, fully equipped and repaired units every 5 minutes.

For now, simply note the presence of the *paused yes* and *activate? 100* attributes for the pulser. We'll come back to those in the section "ON DEMAND SPAWNING"

ON-DEMAND SPAWNING (CLASSIC ME→DML)

This is for the SAMs, Hogs/Sharks and Convoy. The cloners are first activated by standard ME triggers:



As soon as one of the two aircraft are in the game, the relevant flag(s) are set, and the cloner(s) sense that they should spawn. After that, they re-spawn as described under ENDLESS SPAWNS. Above works well with the convoy, and the Air-to-Air groups

Now, with the SAMs, we hit a snag: the SAMs are using a pulser to refresh them regularly, and we don't want the refresh to start unless the SAMs are present (otherwise they would appear after the first refresh cycle after 5 minutes).

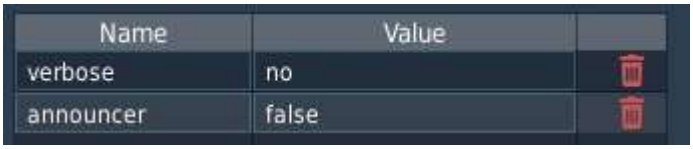
One possible (there are many others) solution is to pause the pulser (*paused yes*). This prevents the pulser from starting any pulses. When the SAM Hunters appear, ME sets flag 100 to one, which is detected by the pulser's *activate? 100* input. This activates the pulser, which causes it to immediately send its initial pulse. That initial pulse causes the SAMs to spawn.



And that is how we put it into this mission: when the SAM Hunters appear, flag 100 is changed, which activates the pulser's initial pulse. The pulser is connected to the SAM's *spawn?* input, causing it to spawn immediately, and from then on, every 5 minutes when the pulser sends out another pulse.

MISSILE PROTECTION

This is quite easy: we add guardian angel to the mission, and then add a config zone that



- Turns off verbose
- Turns off announcer

and leaves all other defaults. Turning off the announcer simply leaves the main guardian angel functionality (remove missiles when they come too close) but removes the breathless commentary.

5.26.3 Discussion

A fully-fledged mission with no Lua.

To make this mission more interesting, change Guardian Angel's configuration to not destroy the missiles, and disable respawning for the SAMs.

5.27 CSAR of Georgia.miz

5.27.1 Demonstration Goals

This mission demonstrates how to use CSAR Manager to easily create and trigger CSAR missions.

Note: due to the nature of CSAR Missions, you must use a Troop-Transport capable helicopter (Huey, Hip, or Hind), as the UI will not respond otherwise.



CSAR Missions can automatically provide pilots with directions to their mission targets, and allows to winch-rescue evacuees that have been grounded on difficult terrain, as well as smoke-mark the LZ. Downed pilots are equipped with an ELT that a player's helicopter can home in on. Note that picking up downed pilots increases your helicopters total weight, so be careful when planning multi-pick-up CSAR missions at high altitude.

5.27.2 What To Explore

5.27.2.1 In Mission

Start the mission in the SU-25T (i.e. *not* a helicopter). Go to communication→Other.. and note that there is only one item available – the one to trigger a new CSAR mission. Even if you trigger this mission, it's not available to you in a fixed-wing plane.

Now change role to one of the helicopters (we'll choose the Huey here, but the others will react the same)

Go to communication→Other… and note that there is a new item CSAR Missions… available. Choose it. This is the CSAR Manager's UI and offers the following options:

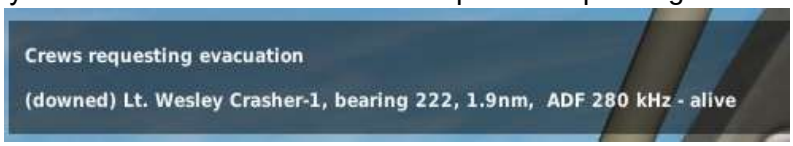
- *List active CSAR*
  This shows a list of all currently available (waiting for pick-up) CSAR missions, along with their ELT's frequency. If vectoring is enabled (by default), this also shows range and bearing to each of the evacuees.

- *Status of rescued crew aboard*
  Shows the medical status of each evacuee that you have picked up. This is meant for

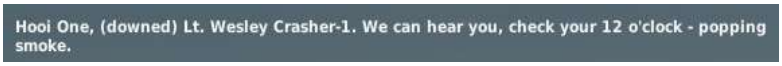
later extension when time-critical missions are supported.

- *Unload one evacuee*
  CSAR Manager automatically loads evacuees when you land close to them. This can overload your helicopter (for example if you are performing a rescue operation in a mountainous region), so you can unload already picked up evacuees one by one and rescue them later.

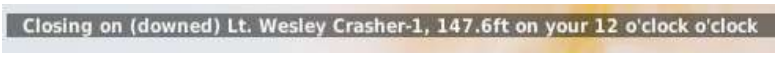
Choose “List active CSAR”, and you will receive the news that one pilot is requesting extraction: a Lt. Wesley Crasher. They are some 1.9 miles away at bearing 222, and their ELT can be located at a certain frequency (here 280 KHz). This frequency can change, as this particular CSAR is set up with randomized frequency. If you are so inclined, set your ADF to this frequency, and listen to the ELT’s shriek. Now check the map and since all units are visible, you’ll also see the downed pilot to the south.



When you get close enough, you’ll receive a new message. It gives you the ‘clock position’ of the LZ, and if enabled, the LZ is marked with colored smoke to help you visually identify the target.



When you get closer still, CSAR manager’s ‘CSAR Chief talk-down’ kicks in that constantly updates your location relative to the evacuee.



If you touch down close enough, the evacuee is automatically loaded into the helicopter. If you hover over the evacuee at the correct height and distance, your crew attempts a winch rescue, and a count-down is initiated. When the count-down reaches zero, the evacuee is safely hooked and automatically winched aboard.



Now return to the airport, and land close to the two fire engines where you originally departed. As soon as you touch down, the CSAR mission concludes successfully. Congratulations!

But… as soon as you touch down with your first evacuee, you receive a new distress call. Before we go off saving this pilot, go to Communication→Other→Start CSAR Mission via Flag 100. Notice the message that appears. Request an update on all current CSAR missions and note that there are now two CSAR missions available. Switch to map view, and you’ll see their locations. You can now rescue these pilots as well. Note that each time that you rescue a pilot, a new CSAR mission is created some 2 nm to the southeast of Senaki-Kolkhi.

If you have the modules installed (and are able to pilot them), switch to other helicopters and see that CSAR manager works with them as well.

5.27.2.2 ME

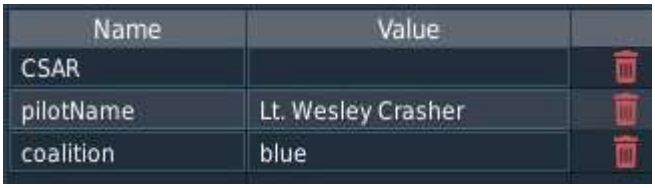
The first thing to notice is that all CSAR GUI is self-contained in the module, so there is no set-up for the mission designer other than importing the module.

For CSAR Manager to work, it requires two important items: the Base (or bases, as CSAR Manager happily accepts multiple bases) where you return downed pilots / evacuees to, and the evacuees themselves.

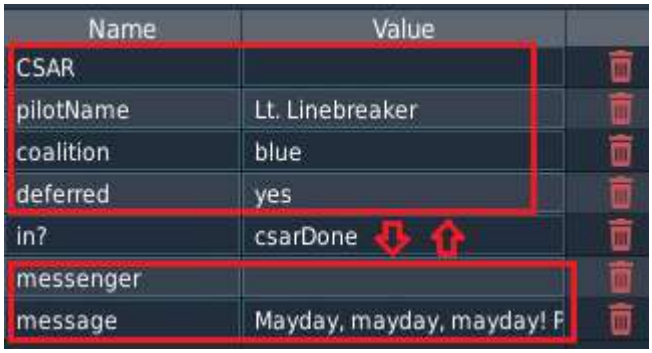


Turn your attention to Senaki-Kolkhi. And note the light blue quad-based trigger zone “CSAR safe”. It only has a single attribute: CSARSAFE, which marks the entire contents of that trigger zone as an area that, if you touch down with your helicopter inside it, and have evacuees loaded, their CSAR missions complete successfully. Whenever you include the CSAR Manager in a mission, remember to place at least one CSARSAFE zone, or the module will complain bitterly at mission start. We chose a quad-based trigger zone only to show that this function also works with quad zones.

Now let’s look at the CSAR Missions (downed pilots). These missions can be created in various ways:



- With a CSAR zone. When you use the basic CSAR Zone, it creates a CSAR mission for the appropriate side (or neutral if none given, neutrals can be rescued by any faction) at the center of the zone when the mission starts up. The very first CSAR mission where you picked up Wesley Crasher is such a mission.

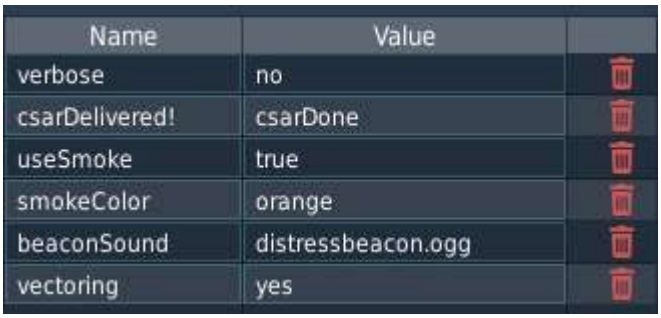


- With a deferred CSAR Zone. These are like the CSAR zone above with an additional important distinction: they do not create their CSAR Mission when the main mission begins, but wait for the flag startCSAR (or one of its synonyms) to change, and then create a CSAR Mission. Since they are controlled by flags, it’s possible to create multiple CSAR missions, as the endlessly repeating CSAR mission featuring Lt. Linebreaker and his clones show. If you use such a configuration, you would usually stack some other modules on top of the CSAR for additional functionality. In our example we stack a messenger onto the CSAR, and connect both their input flags to their common synonym “in?”

- A CSAR can also be initiated from Lua code (not demonstrated here)

- A CSAR can be initiated from other DML modules, for example “Limited Airframes” where player ejections can result in new CSAR Missions (when they eject safely and touch down on land). This is also not demonstrated here

Some important functionality of CSAR Manager is accessible via its configuration zone, and that is what we used to implement the ‘never-ending CSAR’ mission:



CSAR Manager supports a number of convenient flag banging abilities when a player returns a pilot to a CSARBASE: one for red (“csarRedDelivered!”), one for blue (“csarBlueDelivered!”), and one for any successful CSAR (“csarDelivered!”). We use the latter, and wire it into Accident Site-3’s “in?” flag. So every time a player safely delivers a pilot to base, a new CSAR mission is created at Accident Site 3.

5.27.3 Discussion

Again - No Lua!

Since this module’s config zone offers so many cool features, let’s experiment a little:

- Try a CSAR mission with no vectoring. Turn it off, and then see if you can still locate the crash victim
- To make it even harder, turn off smoke
- Change the beacon sound to something you like better

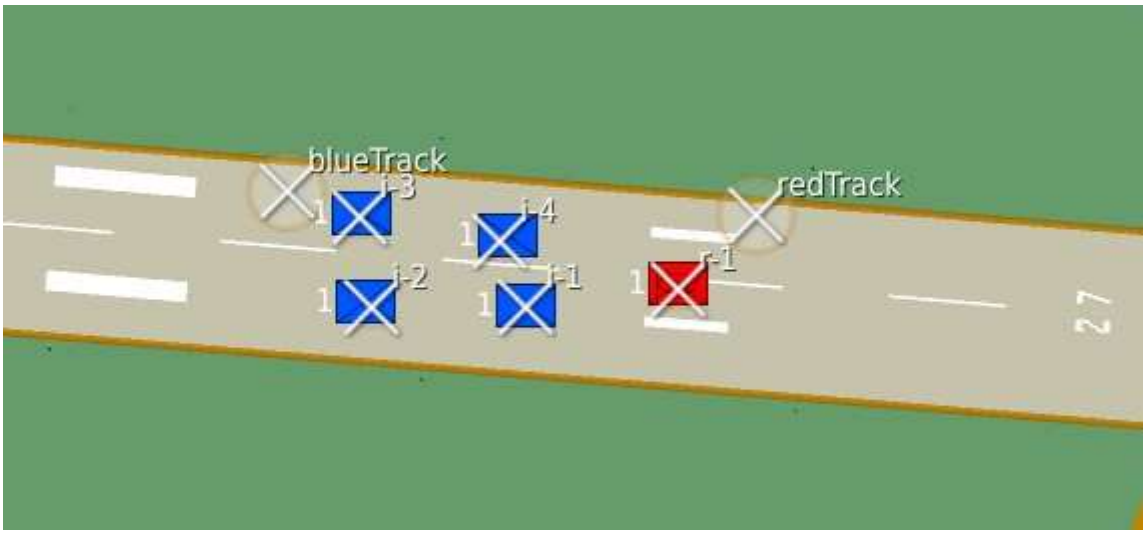
And now for a real challenge: Have your mission create new random CSAR that whenever you successfully bring back one pilot. To make this more manageable, let’s add the following conditions

- The “random” CSAR missions aren’t random by themselves. They are merely pre-placed CSAR missions, and you randomly pick one from all the available ones
- Add a delay of 10 seconds between successful delivery of an evacuee and when a new CSAR mission is generated
- When a new CSAR mission is generated, have it display a message.

5.28 Track This! (Group Tracker)

5.28.1 Demonstration Goals

This rather abstract mission shows how you can use groupTrackers as counters that drive cloners and messengers.



groupTrackers utilize the duality of DCS flags – their somewhat schizophrenic ability to be used as true/false/change indicator to start something, and the ability to hold a positive number as value that can transport additional meaning.

Accordingly, groupTracker's output flags can convey multiple meanings simultaneously, and we'll use both of them. This mission has very little artistic or gaming value, but it serves nicely to illustrate how incredibly useful groupTrackers are.

5.28.2 What To Explore

5.28.2.1 In Mission

Start the mission, and enter the Frogfoot. Press F7 to watch the red Hummer repeatedly murder the four re-spawning infantry, who continually respawn, and eventually kill the Hummer, only to have the cycle repeat.

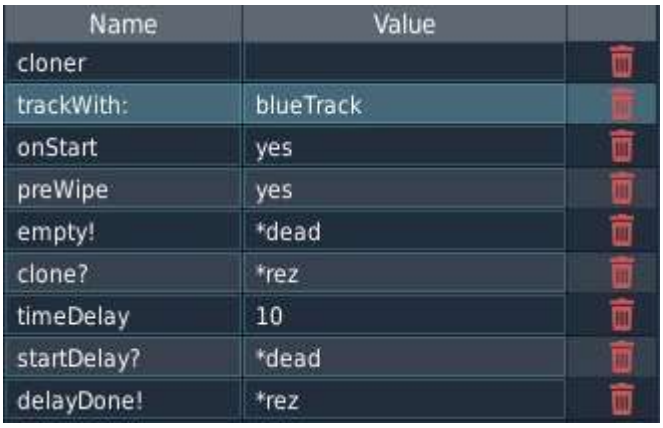
Somewhat more intriguing are the scrolling text messages on the side that accompany the grisly spectacle: a kill count that is updated each time one side makes a kill.



There's very little else to see here, so exit the Frogfoot and head over to ME. On the way over, it may help if you tried to imagine how you would implement this using traditional ME, or even with DML but without groupZones. It's possible, but ain't pretty.

5.28.2.2 ME

The initial set-up holds few surprises: the infantry and Hummer are all built using basically the same stack of modules on the zone:



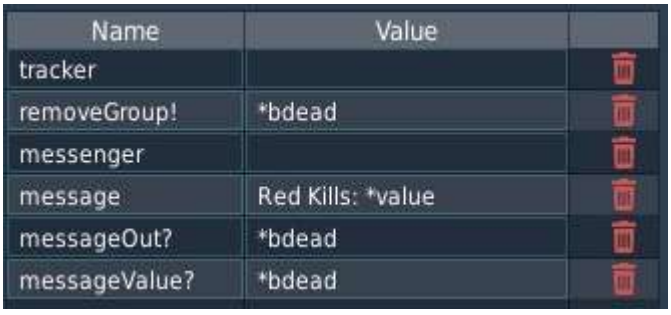
Name	Value
cloner	
trackWith:	blueTrack
onStart	yes
preWipe	yes
empty!	*dead
clone?	*rez
timeDelay	10
startDelay?	*dead
delayDone!	*rez

If we overlook the trackWith: attribute, all we have here is a self-refilling cloner:

- onStart = yes means that we start the mission with a cloned template
- preWipe is set, so prior to a spawn, all remaining units are wiped. This is usually just a cosmetic option and works with some, but not all units
- when the last unit of the last batch is killed, empty! fires on (local) *dead, which is wired into the stacked delayFlag module's startDelay?. It's set to 10 seconds delay (timeDelay = 10), and after the time runs down, the delay bangs! on local *rez, which is wired into the cloner's clone? input, starting the cycle anew.

So far, that relf-repeating cloner is easily understood. The only odd attribute is "trackWith:", which lists the name of another ME Trigger Zone, "blueTrack". So what is "trackWith:"? This is a bit of DML magic, that allows cloneZone to directly talk to groupTracker, and pass the newly spawned groups to that group tracker, allowing it to do what you can't outside of Lua: add groups to a groupTracker while the mission is running (i.e. not at the start of the mission). As DML's main dynamic spawners, cloneZone and spawnZone have their own, privileged onramp to groupTracker, and it's called "trackWith:".

So let's look at what blueTrack is doing:



Name	Value
tracker	
removeGroup!	*bdead
messenger	
message	Red Kills: *value
messageOut?	*bdead
messageValue?	*bdead

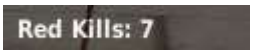
It receives groups of blue infantry whenever they are spawned by a cloner. These groups simply pop into the tracker's watch set and from then on are tracked. Since we have no addGroup! output connected, adding groups results in no action from this tracker.

Eventually, a blue group is killed (since the infantry groups are all single-unit groups, this is rather fast). The tracker notices this and does the following:

- It outdates the new group count. Since no numGroups! output is connected, this results in no further action
- It increases the removeGroup! output (it does this once for each dead group it finds in this pass, usually, that's only one. But if two or more groups die in-between track cheks, the correct amount of increases is done on the removeGroup! flag). This output is wired to the (local) *bdead flag.

So groupTracker actually does very little – at first. It's the way we interpret the information that makes all the difference

Look at the messenger that is stacked on the same zone. It's messageOut? is triggered by the local *bdead flag, which we know increases each time a blue infantry dies. So the death of an infantry triggers the out message



Messenger goes ahead and assembles the message "Red Kills: *value". When it encounters the "*value" wildcard, Messenger then looks at its messageValue? input, which is *also* wired to the same local *bdead flag. Instead of looking for a mere *change* in value, Messenger now loads the current value itself, and exchanges that value for the "*value" wildcard. If that value is now 7 (meaning that this flag got increased 7 times, or 7 blue infantry groups have died in total), the newly assembled (and displayed) text reads "Red Kills: 7"

So, in retrospect, this demonstrates how we can use a single groupTracker outputs in two very different ways on the same zone.

5.28.3 Discussion

Again, no Lua.

And now for some real challenges:

- using the knowledge above, build a group of three cloners that each spawn new single-infantry groups until seven infantry have been killed. At that point, all spawners stop spawning.
- Now do this *without* xFlags

5.29 Watchflags Demo

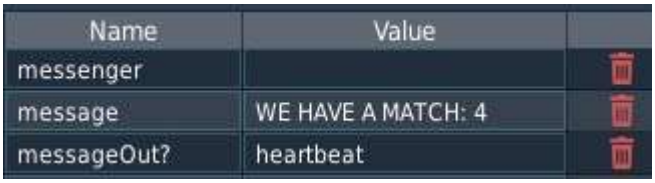
5.29.1 Demonstration Goals

In DML, Watchflags are a central ability, and they can simplify many mission designs by providing on-the-spot condition checking. In other words, Watchflags may well be the secret, unsung heroes of your next mission design.



This mission is designed to merely demonstrate the basic principles behind Watchflags and why they can be so useful.

TRIGGERING A MODULE'S FUNCTION



Many modules can be activated from the 'outside' via an input flag (in the example on the right called 'heartbeat'). This is a flag that the module watches, and – by default – when the **flag's value changes** (e.g. from 2 to 19), the module **interprets this value change as a signal to activate**. The new value is stored, and when the flag's value changes again (say from 19 to 5), that again is interpreted as an activation signal.

This simple mechanism is exceedingly helpful, and enough to satisfy the vast majority of a mission's demands: there mere presence of a plane in a certain zone, for example, may sufficiently fulfill all criteria to advance to the next stage.

Now, in many missions there are situations, that are similar, but require just a bit more sophistication. For example, you may not want to trigger a module when a group was destroyed (which is easily done with a simple trigger cued to the kill event), but when a total of three groups are destroyed. Sure, you can easily build a small automaton (using a count down module) to solve that particular requirement. After designing a number of missions you'll realize that some pattern emerge: quite often, just being able to narrow down the trigger event's change event would greatly simplify mission design.

Now let's look at the fact that we only use a small portion of the information that a flag can carry: Flags carry numbers as information. Right now, by simply detecting a change in the number the flag carries, we use a qualitative aspect to trigger a module. What if we were able to also use some of the quantitative aspects? For example, can we build an input that triggers on the change of flag's number, *but only if that number is now greater than 3*?

And that is what Watchflags are all about: the option to add a small quantitative rule that we apply to a flag in addition to detecting the change. By default, all Watchflags are told to merely detect a change. With a 'triggerMethod' attribute (or its module-specific synonym) we can add one of number of pre-defined additional checks.

In this mission we explore the ability of a Watchflag to detect the following conditions

- change (this is the default behavior of all flags, and the same as if no additional condition was given)
- Increment
- '<(number)': smaller than a number
- '=(number)': equal to a number
- '>(number)': greater to a number
- '=(flag)': equal to another named flag
- '=("flag")': equal to a flag that happens to have a number as name

Remember that there are more conditions that Watchflags can detect, this demo merely scratches the surface.

5.29.2 What To Explore

5.29.2.1 In Mission

Start the mission, enter your trusty Frogfoot. Observe the message that is displayed and follow the instructions (Communication→Other→Start Heart).

A message "pulse started" appears, and every five seconds from now on, new lines of text appear with various content. Just note that they appear, and let this run until you've seen the following message:

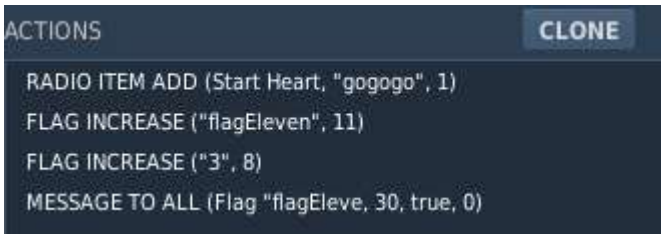


That is the last message we are looking for, and after you have confirmed that the message appears, you can close the mission.

5.29.2.2 ME

This demo explores some of the more common (but not all) rules by using a simple mechanism: a pulser provides an ever-changing, flag value that changes every 5 seconds by incrementing the flag's value by one: "heartbeat". All the messenger zones use the pulser's "heartbeat" flag as input, and process it differently.

However, before we go into details, we need to look at a couple of important settings that are set up during mission start in ME



- First, of course is the radio item 'Start Heart' that, when selected, sets ME flag "gogogo" to 1.
- Then the ME flag "flagEleven" is set to the value 11. This is done via a FLAG INCREASE action simply because, unlike DML, there is no 'Flag Set' action in ME. This only works because we know that at mission start, this flag's value is 0 (zero) and no other action is setting it
- Using the same action, we set the ME flag "3" (that is a flag whose name is "3") to the value of 8

So, when the mission begins, we have also set up two ME flags: 'flagEleven' (which is set to the value 11) and (classic ME) flag "3" (which is set to the value 8)

Now let's look at the pulser "heartbeat". This is offers few surprises:

- It bangs! on the flag 'heartbeat'
- Since pulseStopped is set to yes, the mission starts with this pulser stopped
- It starts pulsing when the flag "gogogo' changes its value
- There are 5 seconds between pulses
- Since no number of pulses are specified, it will pulse until the mission ends
- When pulsing, the output flag ('heartbeat') is always increased by one (pulseMethod = inc)

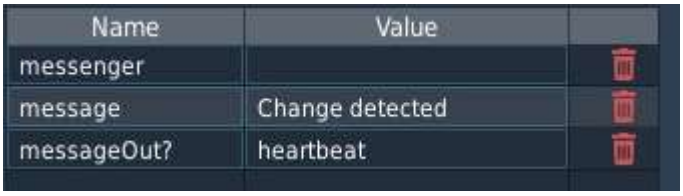


There's also a messenger stacked onto the same zone, which is also wired into 'gogogo' (with messageOut?); the flag that ised used to start the pulser. This is purely cosmetic, as it merely gives you feedback that you have started the pulse.

Now let's look at the remaining messengers that all use differently configured messengers:

Zone "Change"

The 'Change' zone represents the default behaviour (no triggerMethod attribute is given, which defaults to 'change').

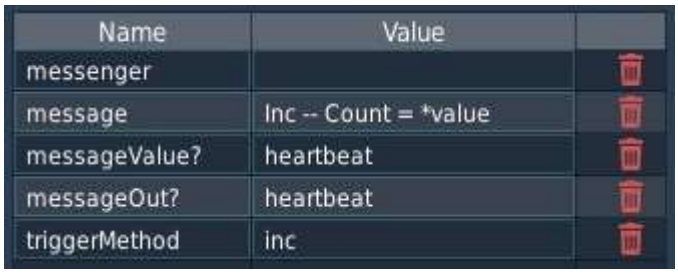


As a result, every time that the pulser changes it's output (the flag 'heartbeat'), this messenger triggers, and writes a new line 'Change detected" to the screen.



Zone "Inc"

This is the first zone with an expanded trigger condition. In addition to having the flag changed, the new value of the watchflag must also be larger than the value that it previously observed (triggerMethod = inc). So in order to trigger, 'heartbeat' must not only change, it must have a higher value than last time messenger checked.



We are also feeding the "heartbeat" flag into messageValue, so it's value can be inserted into the message that we put to the screen. Since the pulser is set to always increase the value of "heartbeat", we are sure that this message appears whenever a pulse is sent, and we now know that we always have the current value on-screen



Zone “Smaller 5”

Here we trigger only as long as the value of heartbeat is less than the number 5 (triggerMethod = <5). Our expectation is therefore that we receive four messages, and when heartbeat has a value of 5 or higher, this module no longer generates output to the screen.

Name	Value
messenger	
message	<5 -- Count = *value
messageValue?	heartbeat
messageOut?	heartbeat
triggerMethod	<5

When you check the game’s output more than 20 seconds after you start the heart, you’ll notice that the line “<5 – Count = …” no longer appears.



Zone “Exactly 4”

This one appears exactly once during the lifetime of our mission: when the value of “heartbeat” is the number 4 (triggerMethod is “=4”)



Name	Value
messenger	
message	WE HAVE A MATCH: 4
messageOut?	heartbeat
triggerMethod	=4

Zone “Six or larger”

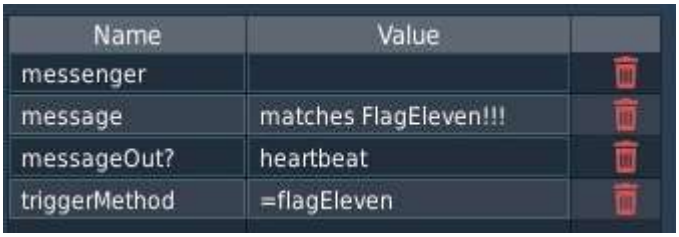
This one may only surprise you initially, due to the way that the zone is named is setting up your expectations. Our goal is to trigger the module when the number of flag ‘heartbeat’ reaches six, and every time from then on (when it is larger than six).

Name	Value
messenger	
message	6 or larger
messageOut?	heartbeat
triggerMethod	>5

However, triggerMethod is set up to “>5”, and – probably because our brains are wired that way – it may take us a few moments to realize that “>5” and “>=6” are logically equivalent (at least when dealing with integers as we are here). DML only supports = (equal), < (less than) and > (greater than), so we’ll have to use >5 as it’s logical equivalent to “six or more”.



Zone “flagEleven”



And now we are getting into the deep end of the pool. DML’s Watchflags can not only compare a flag to some fixed number, it can also compare the value of the Watchflag to any other flag. All you need to do is supply the flag’s name instead of a number. This allows you to perform complex decisions right on your zone. Of course, local flags are supported.

Also, remember that Watchflags not only support equality comparison for other flags, but also “<” and “>”.



Zone “QUOTED number-named flag ‘3’ that is 8”



A historical oddity of DCS is that prior to late March 2022, flag names in ME were limited to numbers. That made mission development quite challenging and unforgiving. As a result, there are many existing missions that use flags with numbers as names, and bad habits die hard, so it’s to be expected that there are missions that (still) use flags with names that happen to be numbers.

Which means that DML should support those backwards-oriented souls. But how can DML tell a number from a flag that uses as number as name? In this case, you must surround the name (which is a number) by double quotes. If a Watchflag sees a number in quotes, it will use the number as name for a flag instead of a number value.



5.29.3 Discussion

Again, no Lua.

Try and experiment with the various conditions, then build a zone that announces only the third kill of an entire group.

And now try this without Watchflags. 😊

5.30 Viper with a double youu (Wiper)

5.30.1 Demonstration Goals

Wiper is a module that removes a variety of objects from the game. This demo runs through it's most important uses, and shows some of its limitations.

5.30.2 What To Explore

5.30.2.1 In Mission

Enter the Frogfoot, press F2, pan and turn the camera until you have a good view of the "Fuchs" ground vehicle. Choose Communication→Other→Start Infantry Respawn Demo. Observe the gruesome spectacle of the Infantry spawning, being killed, and disappearing just before re-spawning. Notice that after each kill, the entire body is removed.



Now press F7 until have a clear view of the Leopard 2 tank. Now choose Communication→Other→Start Tank Respawn Demo. Observe similar carnage, but this time, after each kill, debris, fire and smoke remain.

Re-start the mission and return to your Frogfoot. Press F7 until you have a clear view of the infantry standing next to a large lamp post.



Now choose Communication→Other→Wipe Lamp.

Observe how the lamp does **not** disappear, but the big, curved hangar dome does disappear! Soldier and (static) airfield machinery remain, though. Also note the block of text that has appeared, listing some names next to 'Cat = 1' through 'Cat = 6'. Take note of these names, as we will return to them in ME later



Restart the mission, and again enter your Frogfoot's cockpit.

Hit F7 until you are viewing the Infantry named Solitary" who stands in front of a boring, nondescript building.

Now choose Communication→Other→Wipe Infrastructure. Note how the building, the trees next to it, and the bunker to lot 14 vanish, but the soldier remains.



Finally, choose F7 until you are viewing the single soldier "Nobody", standing next to a tree. Choose Communication→Other→Wipe Tree, and wait a few seconds.

Yup, the tree doesn't budge.

So, let's jump into ME to see what we are looking at

5.30.2.2 ME

This mission is more about the limitations of wiper than its abilities. Wiper is universally and easily understood: you want an object gone, you call the wiper.

Unfortunately, as we'll see, DCS throws a few spanners into the works, so it's important to know about the limitations, and how we can ensure to not waste valuable time when something happens differently from how we expected it.

THE HAPPY CASE

Luckily, wiper works flawlessly when dealing with units and static objects (i.e. all objects that mission designers place by themselves). Removing these works without fail. Use wiper whenever you want to remove objects that you no longer need, or to enable obstructions that you placed yourself in ME. For example, you can use wipers to gradually over time clear containers from a yard until there is enough space for a helicopter to land, etc.

For fine control, use a wiper's wiperNamed attribute to selectively remove objects from a larger group of possible wipe targets.

WIPING REMAINS / WRECKS

As we have seen in the Infantry Respawn demo, wiping can also work clearing the remains from dead units, so they work well with cloned infantry. But unfortunately, DCS can and will deposit objects like wrecks, smoke and flame that currently can't be wiped because they aren't accessible to DML or other script-based tools – as demonstrated by the Tank respawn demo

Name	Value	
cloner		
clone?	infclone	
empty!	infclone	
preWipe	yes	
wipe?	infclone	
wipeNamed	Grou*	

WIPING SCENERY

It gets ugly when you try to wipe scenery. Wiper allows you to try, and will happily remove all scenery objects it finds, but DCS is not giving in willingly.

- Trees usually can't be wiped. Some trees, on the other hand, are parts of nearby scenery objects and hence can be wiped, but not individually without the other scenery object they come with
- As the "Wipe Lamp" and "Wipe Infrastructure" examples show, some clearly visible scenery objects simply can't be wiped.
- You cannot (at least as of this writing) easily deduce which scenery items can be wiped.
- Worse, wiping scenery objects in a zone may have knock-on effects and wipe other scenery that are outside the zone, so always verify which scenery objects are inside a zone with the Inventory attribute during debugging, and then restrict the wipe by supplying a "wipeNamed" attribute and list the objects that you do want removed.



5.30.3 Discussion

Wiper has another ability that comes in handy when dealing with scenery or heavily populated areas.



Wiper can filter the objects that should be removed by name with wildcard. This works well as long as the names are sufficiently dissimilar. It is, for example, easy to remove all objects that belong to a group since they are (when done correctly) all have similar names in-group, and can be differentiated from other groups by their name.

When dealing with scenery, however, many objects have very similar names, as they often are 9-digit object ID that all differ only by one or two digits when close together





In Senaki-Kolkhi, for example there many bunkers close together. Imagine you want to remove only the four upper 'leaves'. When you place a wiper zone, no matter how small, it is near impossible to find a location that correctly removes the four bunkers, but leaves the

other surrounding bunkers intact. The reason is that, as mentioned, DCS's world.search() method is imprecise returning too many objects.

To resolve this, use the following approach. First, during testing, run the wiper with the inventory option to see which objects are returned.



Then, use ME's "*Assign as…*" to identify which objects are the ones that you are interested in, and then list them in the wiper's *wipeNamed* attribute – remember that this attribute supports lists.

The demo mission has a final communication command: Wipe Hangar Top. This removes the four hangar tops simply by listing all objects names that should be removed in the *wipeNamed* attribute.



5.31 Radio Go Go (Radio Trigger)

5.31.1 Demonstration Goals



The radio trigger module provides some comfort 'glue' for mission designers who use DML and with to integrate ME-style RADIO ITEM ADD-based flags. The standard issue here is that ME only supports setting a flag to a single value, and DML usually triggers on a flag change. This means that subsequent selections of the radio menu simply set the same flag to the same value, not triggering any flag.

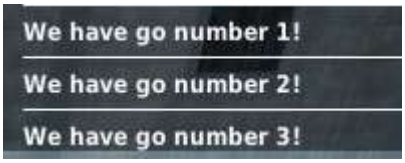
To remedy this, the Radio Trigger module provides a way that notices a flag change in the input, generates a signal on the out! flag, and then resets the input flag so that when the player chooses it again, a new signal is generated.

5.31.2 What To Explore

5.31.2.1 In Mission

Enter the Su-25T and enjoy a tranquil look at Batumi's main airport facilities.

Then choose Communication→Other→Trigger CH1.



Notice that a message appears after a short while.

Choose Communication→Other→Trigger CH1 again.

Notice that again a message appears, slightly modified (it counts the number of times it has been invoked)

You can repeat this as many times as you like, each time a message appears with an increased count.

5.31.2.2 ME

There is very little surprise here: the module watches flag "ch1" for a change, and as soon as it is triggered, bangs on the zone-local "msg" flag that starts the messenger before re-setting ch1.

Name	Value	
radio?	ch1	
rtOut!	*msg	
messenger?	*msg	
message	We have go number <v>!	
messageValue?	*msg	

Since messenger is configured to trigger on change, it would not trigger a second or third message when the item is selected from the radio menu: ch1 is already set to 1.

That's where the radio trigger comes in. As soon as it triggers, it resets the input that triggered it so another activation of the flag via the radio menu is possible.

In our example, we trigger the messenger with radioTrigger's output, which is set to increment (default method out). The message itself also displays the value of the message out flag so that we can easily count how many times the radio item was activated.

5.31.3 Discussion

So you are looking at the radioTrigger module and think – hey, I have an idea: an automatically resetting flag; that's *exactly* what I can use for my next project! I create a flag, feed it into a radioTrigger, and it resets itself automatically, no need for a separate output flag, I merely read the input flag, and it resets itself, right?

The idea is good, but unfortunately it doesn't work. At all. Let's try it. Simply copy and paste the "Radio Check" trigger zone. Now you have two modules that both are looking at ch1, and both should then output a message when the menu item is selected.

Run the mission, and choose the radio item.

Only one message.

Why? Because as soon as one of the two radioTriggers detects the change, it triggers its output, and then immediately resets the input flag. When the other radioTrigger module looks at the input flag, it's looking at the reset flag value, and therefore cannot detect a difference, and does not trigger.

But it gets worse: there is no way to tell beforehand which of the two radioTriggers gets to look at the flag first, and you have therefore just set up a race condition between the two modules.

Now, the same would happen if you used radioTrigger to simply reset the flag and connected another module to the flag to trigger: you'd never know if radioTrigger got to the flag first and re-set it before the other module could trigger. It may also cause the module trigger twice if radioTrigger read it after the first module, and then reset the trigger.

So whenever you are using radioTrigger, make sure it is the only module reading that particular flag; if you need more than one signals generated from that, simply use multiple outputs.

5.32 xFlags – Field Day (Decisions, Flag Testing)

5.32.1 Demonstration Goals

xFlags is an incredibly versatile module that you can use to create complex trigger rules with a snap. This demo shows the different xFlag “requirements” and how they can be used to detect certain common situations. In the demo three Tigrs perform a ‘race’ and we use xFlags to detect various conditions, such as ‘more than one across the finish line’ etc.



5.32.2 What To Explore

5.32.2.1 In Mission

Start the mission and enter your trusty Frogfoot’s cockpit. Once seated, press F7 a couple of times until you are viewing the infantry soldier and his yellow-jacketed buddy. They and the windsock on the other side are the finish line.

Look to the left and notice the three Tigrs. This race isn’t going to be close, it’s a complete set-up. The question is not who will win, but how we detect various situations that can arise in a game with a simple xFlag zone.

Go to Communications→Other…→Start Vehicles

The Race… begins. The Tigrs start rumbling towards the finish line, and as they cross it, multiple messages appear. Take note of what they say, and once all race cars have switched off their engines, you can exit the mission

5.32.2.2 ME

Each of the red trigger zones represents a classic game decision / flag testing situation that xFlags resolve. Each of these zones has a messenger attached that is triggered by that zone’s xFlag.

The setup is simple: there are three target zones (Arrival 1-3) set up behind the ‘finish line’ that each trigger their own (global) flag when a vehicle enters it (using unitZone): oneA, twoA, threeA. When a vehicle enters a zone and triggers the flag, a message is also put to the screen.





The red xFlags zones each take these flags as input, and using different requirements, will trigger at different times. Let’s go though these one by one and see how they work

One Of Us

This xFlag should fire when the first of the “race cars” (“us”) has entered its target zone. Unlike in this demo mission, we don’t really know which of the three will win, so this decision mimics the requirement that from a set of flags, at least one (any) is sufficient to trigger the output flag. Note that the flag will also trigger if more than one input flags were true, but since an xFlags stops checking after it has fired, this situation does not arise.

Name	Value	
xFlags?	oneA, twoA, threeA	
require	any	
xSuccess!	*hit	

All Of Us

This xFlags fires when all three input flags have triggered. It will patiently wait while one vehicle after the other trundles over the finish line and trigger their flags. Only after the third has arrived do we see this xFlag trigger

Name	Value	
xFlags?	oneA, twoA, threeA	
require	all	
xSuccess!	*hit	

Exactly Two Of Us

This xFlag will only fire if two (no more no less) of the input flags have triggered. It would not fire if, for example, all three were triggered. This

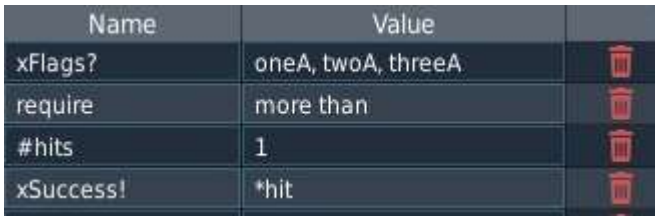
Name	Value	
xFlags?	oneA, twoA, threeA	
require	exactly	
#hits	2	
xSuccess!	*hit	

is in contrast to ‘One of us’ that merely requires a minimum, while exactly requires an exact match.

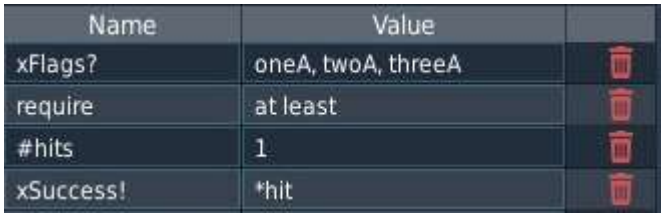
You control which number to match with the #hits attribute (remember that #hits can also be another flag – simply pass that flag’s name. Don’t forget the double quotes if it is a numbered flag)

More Than One Of Us
Triggers xFlag when at least two (or more) flags have triggered upon inspection.

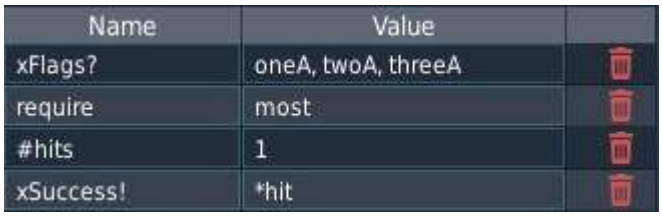
Note that again, #hits carries the information how many flags should be triggered at minimum.



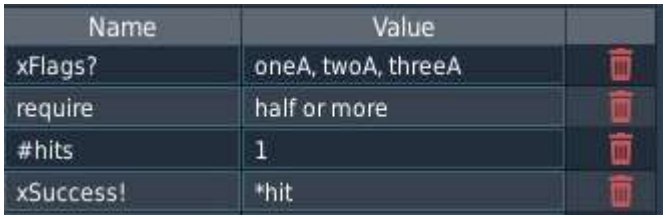
At Least One Of Us
This requires that at least one flag has triggered. It is different to One Of Us in that #hits controls the minimum, and is therefore a more specific version of ‘some’.



Most Of Us
When you need more than half of the flags to have triggered. Note that if exactly half have fired (e.g. 2 from 4), this will not trigger the xFlag.



Half Or More Of Us
The pendant to Most Of Us if you also want to trigger the xFlag when exactly half or more of all the flags have triggered.



5.32.3 Discussion

Resetting One Of Us (Advanced Topic)
There is another zone that demonstrates how an xFlag with reset works. Here we short-circuit the xSuccess! Flag into xReset?, which will reset the entire xFlag and load the current flag state of all input flags as new zero state.

During the race, this causes the xFlag to trigger each time a car passes the finish line (or rather, enters it's unitZone, which then triggers its arrival flag.

Name	Value	
xFlags?	oneA, twoA, threeA	
require	any	
xReset?	*hit	
xSuccess!	*hit	

You may wonder why this doesn't immediately re-trigger, since we know that the first car has arrived.

The answer is that the triggered flag now has been integrated into the xFlag's zero state, and since the watchflags trigger on change, that formerly triggering value now has become the baseline and will no longer trigger that flag.



Why will it no longer detect the fact that the first is triggered?

Zone-Local Verbosity

DML uses the 'verbose' attribute in the module's config zones to switch on a debug mode for that module. They become 'verbose' outputting a lot of information on the screen to help you debug your mission.

Some modules also support zone-local verbosity, meaning that only the modules in that zone (and only those who support zone-local verbosity) turn on their debug mode. xFlags supports local-zone verbosity.

verbose	yes	

Add a 'verbose = yes' attribute to one of the xFlag zones to test this ability. Be careful to remove these verbose flags afterwards because they can litter your mission with messages, and that usually happens at the worst time imaginable.

The DML Quick Reference tells you which modules currently support zone-local verbosity.

5.33 Virgin (Civ) Air / Air Caucasus II / One-Way Air (CivAir)

5.33.1 Demonstration Goals

CivAir is a high-performance drop-in module to generate civilian air traffic. It functions out of the box, and can be easily customized. It's mostly used to make a mission seem more lively, but can spice up a mission by providing aircraft that must not be shot down.



5.33.2 What To Explore

There are two missions here to explore: Virgin (Civ) Air, and Air Caucasus II. We'll discuss both.

5.33.2.1 In Mission

Virgin (Civ) Air is a blank mission with a single player Frogfoot to enter (cold), and CivAir as it functions out of the box. Enter the mission, and press F2 multiple times to step through the airborne flights. After a while stepping through the planes, you'll notice additional civilian flights that sit on the tarmac.

Accelerate time and watch the aircraft take off, fly, and land. Switch to F10 map, and zoom out. The white icons all depict a civilian flight under civAir's control. Note that the flights are fully randomized, and fly all over the Caucasus map. Note also the diversity of airframes.

Now switch to Air Caucasus II, and do essentially the same. Note that now the ten planes all bunch together: they only originate from, or fly to one of the following airfields: Batumi, Kobuleti, Kutaisi, Senaki-Kolkhi and Sukhumi-Babushara. The never fly to any other destination or depart from another airfield.



Also note that there are fewer aircraft types compared to Virgin (Civ) Air: there are only the (very common) Yak-40, Hercules and Antonov An-26 (just ignore the C-17As in the image above, that screenshot was taken from an earlier version of the demo where the Globemaster was in the mix)

5.33.2.2 ME

Virgin (Civ) Air shows how an unmodified drop-in of Civ Air looks in a mission. It's completely hassle-free, truly random, and a nice addition to any mission that has cycles to waste for some atmospheric sugar (CivAir itself uses negligible processing power, the AI planes only very little).



Air Caucasus II demonstrates how to customize civAir in only a few minutes with the following additions to your map:

We add a trigger zone close to the airports of Batumi, Kobuleti, Kutaisi, Senaki-Kolkhi and Sukhumi-Babushara and add just a single attribute: civAir. When civAir starts up, it detects these zones, and builds a new list of airfields that are allowed for civAir from all zones with that attribute. In the Virgin (Civ) Air demo, no such zone was present, and Civ Air then proceeded to assemble all airfields it finds on the map into the list of allowed airfields.

Note that the civAir attribute has no value and is still correctly recognized. As long as the value for this attribute isn't the word 'exclude', civAir uses the closest airfield to that zone as a departure or destination point for a civAir flight.

Name	Value
civAir	

Air Caucasus II uses a config zone to change the air frames that are used. When you look at the planes that are flying in that mission, you'll notice that the Yak-40 small business jet is much more prevalent than the other types. This is because in the aircraftTypes attribute, that type is listed thrice, meaning that it is three times more likely that this type is chosen than the other types that are present (C-130 "Hercules", An-26B).

Name	Value
maxTraffic	10
aircraftTypes	Yak-40, Yak-40, Yak-40, C-13

5.33.3 Discussion

As you can see, civAir works well straight out of the box, and can easily be customized to use only a few airfields and airframes simply by adding some zones and attributes

5.33.3.1 The popular Yak-40

As mentioned above, the Yak-40 is listed three times in the aircraftTypes attribute, making it three times as likely for it to appear than any other plane. Let's find out how that looks. Remove two of the three Yak-40 entries, then run the mission again. Paging through the various planes, you should see that now the distribution is even between the three airframe types. Now also add a C-17A into the mix and see what happens.



5.33.3.2 Making Batumi more popular

So let's make Batumi more popular with the jets. Click on the Trigger Zone "Civ Air Hub-1" next to Batumi Airfield, copy, and paste right next to the first one. Now there are two Trigger Zones with a civAir attribute close to Batumi, making it appear twice in the inclusion list. Run the mission. Note that now flights are more likely to start or end in Batumi, but still there will be no flights from and to Batumi – civAir is smart enough to detect and prevent this.



Note that the list of traffic centers is displayed when you set verbose to true in civAirConfig.

5.33.3.3 Increasing Traffic

Now let's try and stress your CPU and DCS ground control. Set the maxTraffic attribute in civAirConfig to 100. Start the mission. Immediately, it starts up with 50 planes in the air, racing to one of the different airfields.

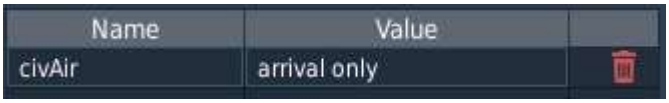
Now, once every 20 seconds (assuming you left ups at 0.05), a new aircraft is added, so it will take almost 17 minutes for the remaining 50 planes to spawn on their various airfields and start their warm-up cycle.

Handling that many planes can become an issue with DCS, as it doesn't handle many planes in congregating on the same airfield well. You may observe strange mutually preventing loops of aircraft in the pattern, or mutually blocking planes on the ground – DCS may even spawn multiple planes over each other, causing some short-lived mayhem until the are all de-spawned (this is a DCS bug – a future version of civAir might contain code to work around this). But the main take-away is that civAir can, and DCS may some day be able to, handle a large number of civilian aircraft without problems, and no discernible performance hit.

5.33.3.4 One-way Air



Another demo (One-Way Air) demonstrates CivAir's ability to 'direct' the flow of aircraft for airfields. Using the values 'departure' or 'arrival' for the CivAir attribute makes that airfield only eligible for take-offs ('departure') or landings ('arrival')



Note that this is compatible with CivAir's concept of multiple zones for the same airfield. Adding either value (arrival/departure) to additional zones for an airfield makes it more likely for a take-off or landing to happen there.

5.34 Count Bases Blue (baseCaptured, xFlags)

5.34.1 Demonstration Goals

This mission demonstrates how we can use the various baseCaptured outputs to trigger other modules:

- The change-of-hands DML flag to trigger a message and
- ownership flag to count (using xFlag's 'lesser' count function) the number of bases in blue's possession.



5.34.2 What To Explore

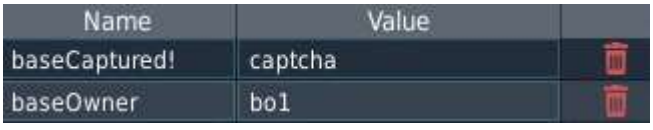
5.34.2.1 In Mission

Enter the mission, and watch how the various Leo tanks approach their target FARPS. After a while, three blue FARPs (from a total of four) are being captured.

Each time a FARP is captured, a message appears with a new total of FARPs that blue still holds.



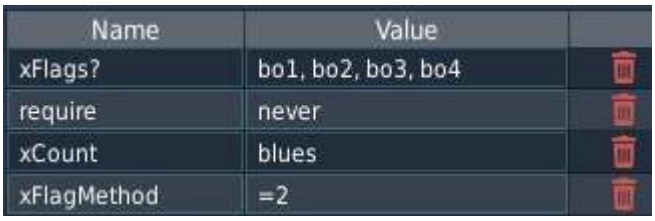
5.34.2.2 ME



Let's begin with the baseCapture zones. They are all set up similarly. When their associated (nearest) base is captured by another faction, they all bang! on the (global) flag captcha. Also, individually, each zone transmits the current owner on the (also global) flag boX, with X being different for each base (so I can easily tell them apart: bo1, bo2, bo3, and bo4).

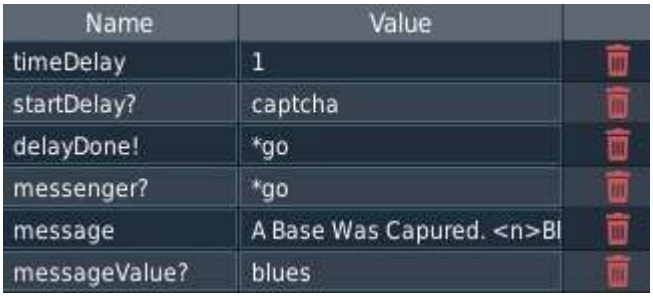
Now, these boX flags are all collected by an xFlags module, configured to function as a counter:

- Input flags are bo1, bo2, bo3, bo4, the ownership flags from the four FARPs. Since they are output from a baseCapture module we know they can only hold three possible values: 0 (neutral), 1 (red), 2 (blue)



- Require is set to 'never', meaning that the xFlags module will continuously run, and never stop, no matter what configuration bo1-bo4 have. This is important since 'require' defaults to "some", so if we did not change it to 'never', xFlags would stop after the first base is owned by blue.
- xFlagMethod is set to "=2", meaning that xFlags looks for input flags whose value is equal to the number two.
- xCount is an output that is set to the number of 'hits' on the input flags, i.e. the number of input flags that meet the requirement "=2". In other words, xCount counts the number of input flags that are equal to the number two – which happens to be the number of bases that are owned by blue. That value is output on the (global) flag named 'blues'.

And finally, the messaging stack: Two modules that run in sequence: the initial signal 'captcha' (which is the signal that a base was captured, as put out by any of the four bases) comes into a delayFlag's startDelay? input and triggers a delay of one second, only to be put out on the (local) '*go' flag.



The (local) '*go' flag triggers the messenger module which gathers the current value of the 'blues' flag (which is the current number of bases owned by blue, as put out by the xFlags' xCount output) and inserts it into the message instead of the <v> wildcard.



5.34.3 Discussion

This deceptively simple mission uses a number of tricks to accomplish what it does. Let's walk through them one by one.

Counting blue bases

We use xFlags' ability to count (and publish via the xCount output) the number of input flags that meet the requirement '=2', i.e. the number of input flags that equal the number two. Since the input flags are the raw 'owner' output from baseCaptured, a flag that is qual to two means that it comes from a base that is owned by blue.

'Never' ever ever

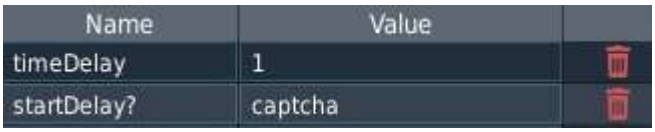
So why did we set 'require' to 'never'? Remember that xFlags pauses once the requirement is met, and dafaults to 'some'. In default configuration, the first time that a base is owned by blue – which is the condition the mission begins at. Since always want xFlags to supply us with the number of bases owned by blue, we tell it to never stop – hence the requirement.

Counting blue – revisited: Base ownership versus base loss.
So why are we using an xFlags module to count the number of bases owned by blue in such a roundabout way? Couldn't we just use a capture event as indicator that blue lost a base, set 'dec' as method for baseCaptured! DML method, and start with 4 as value for 'blues'?

Yes, but that only works when we make a big assumption: that blue can never re-gain any of it's bases. Because if it did, 'blues' would no longer correctly reflect the number of bases blue holds. If read and blue ownership traded the same base a few times, *the value of 'blues' could easily reach zero with blue still holding on to three bases*.

So if you, for example, made it a win condition that red should capture more than 3 blue bases, simply counting capture events is not sufficient if blue can re-capture bases.

Waiting… for what exactly?
In our exploration we have glossed over this bit: the messenger module is triggered after a time delay of one second – i.e. the message is displayed one second after a baseCaptured module's message is received. But why?



The answer shines a light on a slightly ugly facet of DML's inner workings: all modules run on their own timer, and there is no way to tell when which module checks their input. In other words, there is no way of telling the order in which they process the flags. And this can be a problem because for the messenger module, this can introduce a race condition: it could, conceivably process a baseCaptured signal before xFlags does (there's a roughly 50/50 chance of this happening). In this case, however, the messenger module prints the message to the screen before xFlags has had an opportunity to update the 'blues' flag, which carries the number of bases owned by blue. The result is that the messenger displays an incorrect base count for blue.

The time delay of one second changes this, as it forces a time delay that guarantees that xFlags has completed its own processing. But why 1 second? Because that is the value that most modules have their ups (updates per second) set to by default. If you ever run into a possible race condition, you can often resolve this with a time delay for the signal for the module that should run last, and the delay should be at least as long as the ups for all modules that contribute (sometimes, when there are multiple cascades of modules involved, you may have to add multiples of that to account for all cascades).

Challenges:

Can you change the mission in a way that:

1. outputs a message 'Red Won' when red owns three bases?
2. without using the value of the captcha flag?

5.35 DML Mission Template.miz – (Lua Only)

5.35.1 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

5.35.2 What to explore

5.35.2.1 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, not the lines of text on the right side. The 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.

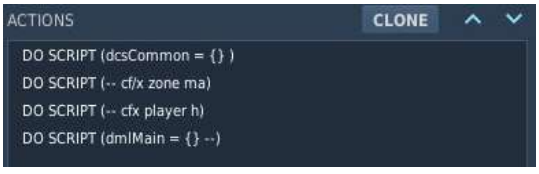
Keep an eye on the right side and now change plane slots, eject, crash the plane, and watch other planes start-up etc. As time progresses, more events are chronicled.

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.

5.35.2.2 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 dmlMail 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 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 "dmlMainConfig"** 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



- 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 dmlMain 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 dmlMainConfig are reflected when the mission runs.

- Note that you can add your own aircraft and they are all automatically supported

5.35.3 Discussion

dmlMain 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!

5.35.3.1 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
```

5.35.3.2 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.

5.35.3.3 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.

5.35.3.4 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.

5.35.3.5 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
```

5.36 Landing Counter.miz – (Lua Only)

5.36.1 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.

5.36.2 What To Explore

5.36.2.1 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.

5.36.2.2 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

5.36.3 Discussion

5.36.3.1 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.

5.36.3.2 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

5.36.3.3 Relevant Code
So, lets 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
```

5.36.3.4 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.

5.37 Event Monitor.miz (tbc)

5.37.1 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.

5.37.2 What To Explore

5.37.2.1 In Mission

Start the mission and, jump into your favorite plane and cause the events that you are looking for.



5.37.2.2 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).

5.37.3 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

5.38 Mission.miz

5.38.1 Demonstration Goals

5.38.2 What To Explore

5.38.2.1 In Mission

5.38.2.2 ME

5.38.3 Discussion

6 Mission Design FAQ

Got a question? Check here first!

6.1 General

6.1.1 Loading Modules

- *How do I load modules?*
  There's a video available, and it's also explained in this manual.

- *Do I have to load DML at mission start?*
  No, but it is the best point in time to do so. Some modules assume that they load at mission start, and the information they gather from the original mission may not match the current situation if you do not.

- *Is the order in which modules load relevant?*
  Yes, in that you need to load 'dependent' modules before you load that module. For example, you must load dcsCommon and cfxZones before you load Messenger.

6.1.2 Module Misbehaving?

- *My module only reacts the first time or not at all…?*
  Triggers react on change of the flag's value. Make sure that you are *triggering* with a DML flag method like 'inc' (one that will change the flag's value each time) if you want the module react every time. Specifically, be aware that the 'flip' method works well to guarantee a change only if one module changes that flag – if two or more can change the flag, if they fire at the same time, two flips in a row will reset the flag to its original value.
  On the other hand, your Watchflag condition may be set incorrectly, so also ensure that it set to 'change'

6.2 Module Specific

6.2.1 Messenger

- *Message does not appear although it triggers correctly*
  Check the coalition attribute. It defaults to 0 (all), but if you are stacking a messenger with other modules on the zone (e.g. unitZone), ensure that any coalition attribute present uses a synonym that does not conflict with messenger.

6.2.2 Object Destruct Detector

- *Module used to work, but suddenly has stopped working in my mission*
  From release to release, DCS might change the objectID of some buildings/objects on the map. Verify that the objectID matches the one you are looking for

6.2.3 SSBClient

- *I have added and configured SSBClient, but it does not block any slots*
  SSBClient only works when the mission runs in multiplayer. Start it as a self-hosted Multiplayer game

- *Hey, I did the above, still nothing!*
  SSBClient also needs SSB installed in your Server Hooks folder.